National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

NetKernel: A Net-Oriented Paradigm for Parallel Architectures

by

Theodore John Becker ⓒ

A thesis submitted to the Faculty of Graduate Studies and Research in partial fufillment of the requirements for the degree of Masters of Science.

Department of Electrical Engineering

Edmonton, Alberta
Spring 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-82202-3

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Theodore John Becker

TITLE OF THESIS: NetKernel: A Net-Oriented Paradigm for Parallel Architectures

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Theodore John Becker

Theodore John Becker
4016 Vardell Rd. N.W.
Calgary, AB, Canada
T3A 0C4

Feb 26, 1993

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **NetKernel: A Net-Oriented Paradigm for Parallel Architectures** submitted by **Theodore John Becker** in partial fufillment of the requirements for the degree of Masters of Science.

Prof. W. Joerg (Supervisor)

Don Koval
Dr. D. Koval

J Schaeffer
Dr. J. Schaeffer

Date: Feb 24 1993

Abstract

NetKernel is an experimental kernel for concurrent programming using parallel hardware. It provides low level support for net-oriented programming paradigms. It is intended to provide a testbed for research into mapping of concurrent software. The basic requirement for NetKernel is to flexibly support the mapping of software networks to physical networks without any modification of source code.

A NetKernel program is formulated as a software network of interconnected processes which communicate by asynchronous message passing. A software network is mapped to a physical network for execution. NetKernel uses separate descriptions for these three fundamental concepts: software network, physical network, and mapping. This facilitates experimentation with different mappings and different configurations of hardware. The kernel runs on each station in the physical network, managing the execution of concurrent processes and providing a distributed message system. The kernel also includes facilities for loading, controlling, and monitoring the execution of processes at stations throughout the physical network. NetKernel has a microkernel design, consisting of a core which provides process switching and local interprocess communication. A collection of kernel processes provides additional services including transparent nonlocal communication. Development tools running on host workstations provide an environment for loading, monitoring, and controlling the execution of software networks executing on a physical network of target processors. Where possible, complexity has been shifted from the targets to the hosts. This allows NetKernel to run on target systems with very little memory.

Our current hardware and implementation do not have particularly high performance. The intention is to provide an economical platform for the investigation of novel concepts in concurrent software rather than to develop a high performance parallel computer. The requirement for flexible mapping has been achieved and demonstrated. It is anticipated that NetKernel will next be used to explore a new design notation based on Petri Nets.

Acknowledgments

I thank my supervisor, Professor Werner Joerg, for opening my eyes to the concurrent systems approach as a powerful way to deal with many problems. I am also very grateful to him for providing rapid feedback on my thesis, even during difficult times. Many thanks to my examination committee members, Professor Don Koval and Professor Jonathan Schaeffer, for their time and valuable suggestions concerning this thesis.

I have enjoyed the privilege of working with several undergraduate students who have contributed to various aspects of NetKernel. Thanks to Darryl Maier (Host Interface Program), Collin Roth (Debugger), and fellow graduate student Kal Voruganti (Script Generator). Also to Andrew Jarman, John McDougald, and Thomas Yoon for their work on shared memory hardware.

I especially thank my family for their unwavering support and encouragement throughout my education.

Finally, I gratefully acknowledge the Department of Electrical Engineering, the University of Alberta, and the Natural Sciences and Engineering Research Council for financial and technical support.

Contents

List of Tables
List of Figures
List of Symbols
List of Abbreviations
Trademarks
Chapter 1 Introduction 1
1.1 Concurrent Software 1
1.2 Embedded Systems 1
1.3 Motivation 2
  1.3.1 Parallel Programming Paradigms and Architectures 3
  1.3.2 Message Passing vs. Shared Data 4
  1.3.3 Net-Oriented Programming Paradigms 5
1.4 NetKernel 6
1.5 History of the MPAXS Project 7
1.6 Outline 7
Chapter 2 NetKernel Paradigm and Architecture 9
2.1 Overview 9
2.2 Operational Requirements 10
Part A: Paradigm
2.3 Software Network 10
  2.3.1 Process 10
  2.3.2 Software Network 11
  2.3.3 Provision of Services by Kernel Processes 12
  2.3.4 Input/Output 13
  2.3.5 General Process Models 14
    2.3.5.1 Client-Server 14
    2.3.5.2 Data Flow 15
2.4 Message System 16

2.4.1 General 16
2.4.2 Primitives 16
2.4.3 Messages 18
2.4.4 Process Blocking and Timeouts 18
2.4.5 Transparency and Configurability 19

Part B: Architecture

2.5 Physical Network 20
2.5.1 Stations 20
2.5.2 Links 21
2.5.3 I/O Devices 21
2.5.3.1 Treating Processors as I/O Devices 22
2.5.4 Physical Network 22
2.5.5 Variations in the Environment On Different Station Types 23
2.6 Mapping 24
2.6.1 Primary Mapping 24
2.6.2 Secondary Mapping 24
2.6.3 Example 25
2.6.4 Mapping Objectives 26
2.6.5 Constraints 26
2.7 Software Development 27
2.7.1 Description Files 27
2.7.2 Host Interface Program (HIP) 27
2.7.3 Construction 27
2.7.4 Mapping 28
2.8 Simplicity and Adequacy of the Paradigm and Architecture 28
Chapter 3 NetKernel: Design and Implementation 30
3.1 Overview 30
3.2 Design Philosophy 31
3.3 The MPAXS Hardware 32
3.3.1 Processors 32
3.3.2 Physical Network 33
3.4 High-Level Design 34
3.4.1 Summary of Operation 34
3.4.2 Message System 35
3.4.3 Kernel Processes 36
3.4.4 Microkernel 38

3.4.5 Interface 39
3.5 Microkernel Design 40
3.5.1 Module Summary 40
3.5.2 Kernel Data Structures 41
3.5.2.1 Summary of Structures 42
3.5.2.2 Mutual Exclusion 42
3.5.3 Process Scheduling 43
3.5.3.1 Process State and Control State 43
3.5.3.2 Scheduling Mechanism 44
3.5.4 Dynamic Memory Management 45
3.5.5 Message System 46
3.5.5.1 Send() 46
3.5.5.2 Wait() 47
3.5.5.3 Take() 47
3.5.5.4 Reply() 47
3.5.6 Exception Catcher 48
3.5.6.1 Device Drivers 48
3.5.6.2 Interrupt Servicing Through the Microkernel 49
3.5.6.3 Handler Processes 50
3.5.6.4 Process Classification 51
3.5.7 Clock 51
3.5.7.1 Statistical Profiling 51
3.5.7.2 Runaway Process Suspension 52
3.5.7.3 Updating Station Time and Timers 52
3.5.8 Miscellaneous 53
3.5.8.1 Event Logging 53
3.6 Kernel Processes 53
3.6.1 Interaction Between Kernel Processes 54
3.6.2 Process Decomposition 54
3.6.2.1 Considerations 54
3.6.2.2 Processes Priorities 55
3.6.3 Station Operation 56
3.6.3.1 Process Manager 56
3.6.3.2 Default Exception Handler 58
3.6.3.3 Debugger 58
3.6.3.4 Serial LDD/IODD (IODD role) 59

            3.6.3.5 Operator Interface 60
            3.6.3.6 Profiler 61
            3.6.3.7 S Record Loader 61
            3.6.3.8 Timer 61
            3.6.3.9 Log 62
            3.6.3.10 Init 62
        3.6.4 Distributed Message System 62
            3.6.4.1 Overview 62
            3.6.4.2 Connection Database 65
            3.6.4.3 Router 65
            3.6.4.4 Serial LDD/IODD (LDD role) 66
    3.7 Correctness of Design 68
        3.7.1 Safety 69
        3.7.2 Liveness 69
            3.7.2.1 Deadlock 69
            3.7.2.2 Lockout 70
            3.7.2.3 Termination of System Calls 72
    3.8 Implementation 72
        3.8.1 Compiler and Other Support Software 73
        3.8.2 Code Statistics 73
        3.8.3 Process Priorities 73
        3.8.4 Developing User Processes 74
        3.8.5 NeXT Workstation Prototype 74

Chapter 4 Development Tools: Design and Implementation 75
    4.1 Overview 75
    4.2 Description Files 76
        4.2.1 Software Network Description (SND) 76
        4.2.2 Physical Network Description (PND) 77
        4.2.3 Mapping Description (MD) 77
    4.3 Host Interface Program (HIP) 78
        4.3.1 Motivation 78
        4.3.2 Internal Network Model 79
        4.3.3 Overview of Process Management Commands 80
            4.3.3.1 Starting a Software Network 80
            4.3.3.2 Other Manipulative Commands 81
            4.3.3.3 Status Commands 81

4.3.4 Command Modules 81
4.4 Script Generator (SG) 82
4.5 Event Log Report Generator (ELRG) 83
4.6 Implementation 83

Chapter 5 Related Work **84**
5.1 Overview 84
5.2 Types of Environments which Support Concurrent Processes 84
5.3 Influential Concurrent Programming Paradigms 85
5.4 Contemporary Systems 86
5.4.1 Amoeba 86
5.4.2 CAPER 87
5.4.3 CONIC 87
5.4.4 EMPS 88
5.4.5 Enterprise 88
5.4.6 Transputer Systems 88
5.4.6.1 First Generation 88
5.4.6.2 Parallel C 89
5.4.6.3 TOPS 89
5.4.6.4 T-Rack 89
5.4.6.5 Second Generation 89
5.5 Comparison and Discussion 90

Chapter 6 Conclusions **93**
6.1 Conclusions 93
6.2 Future Work 93
6.2.1 Design Notation Based on Petri Nets 94
6.2.2 Performance Prediction and Mapping 96
6.2.3 Improvements and Extensions to NetKernel 97

Bibliography **100**

Appendix A Glossary **103**

Appendix B Selected Aspects of the NetKernel Interface **105**
B.1 Function Prototypes for Message Passing System Calls 105
B.2 Command Interface of the Process Manager 106
B.3 HIP Commands for Process Management 107

Appendix C Demonstration Programs for NetKernel **109**
C.1 Ring Program and Message System Performance 109

C.2 Synthetic Program with Mapping Examples 111
C.3 Repulse Game 113

Appendix D Example Source Code and Description Files **115**
D.1 SND, PND, MD 115
D.2 slave.c, PTD for slave 116
D.3 Script for HIP Generated by SG 117

Appendix E Event Log Report Showing Clock Servicing **118**

List of Tables

Table 1 MPAXS processors and their classification **32**
Table 2 Summary of kernel processes **37**
Table 3 Summary of kernel data structures **42**
Table 4 Periods of system clock activities **52**
Table 5 Relative priorities of kernel processes **55**
Table 6 Source code statistics **73**
Table 7 Message throughput as measured using a ring of two processes **110**
Table 8 Throughput measurements for various mappings of SYNTH **113**

List of Figures

Figure 1 Process **11**
Figure 2 Example of a software network **12**
Figure 3 Hypothetical software network to monitor a reactor **13**
Figure 4 Examples of client-server relationship between processes **15**
Figure 5 Examples of data flow configuration of processes **15**
Figure 6 Guest process **17**
Figure 7 Processor classification and functions **20**
Figure 8 Link classification and examples **21**
Figure 9 Hypothetical physical network **23**
Figure 10 Example mapping **25**
Figure 11 Hardware of the M68000 target station **32**
Figure 12 Examples of physical networks constructed from MPAXS hardware **33**
Figure 13 High-level views of the kernel **34**
Figure 14 Kernel processes of the M68000 version of NetKernel **36**
Figure 15 Structure chart of microkernel modules showing entry/exit points **41**
Figure 16 State diagrams for process state and control state **44**
Figure 17 Data path and units exchanged for a local message **47**
Figure 18 Process classification and functions **51**
Figure 19 Example of Serial LDD/IODD managing two I/O devices. **59**
Figure 20 Dialogue between a terminal and destination x through Operator Interface **61**
Figure 21 Data path and units exchanged for a nonlocal message **63**
Figure 22 Distributed message system processes at an example station **64**
Figure 23 Some routes through a hypothetical physical network. **65**
Figure 24 Communicating Serial LDD/IODD processes at adjacent stations **67**
Figure 25 State Diagrams for (a) receiver and (b) transmitter **68**
Figure 26 Client-server relationships among kernel processes **70**
Figure 27 High-level design of HIP **82**
Figure 28 Compiling a Petri Net design description into a NetKernel software network **95**
Figure 29 Compiling by collapsing **96**

Figure 30 Software network for a ring consisting of a master and two slaves **109**
Figure 31 Software network for SYNTH **111**
Figure 32 Five mappings of SYNTH to four different physical networks **112**
Figure 33 Software network for Repulse game **114**
Figure 34 Example output from Event Log Report Generator **119**

List of Symbols

$\varnothing$	empty set
$\mathbf{N}$	set of natural numbers 0,1,2,3,...
$\{a \mid P\}$	set of elements satisfying property P
$\in$	is an element of
$\notin$	is not an element of
$\subseteq$	set inclusion
$\cap$	set intersection
$\cup$	set union
$\forall$	for any
$\exists$	there exists
$\lvert E\rvert$	number of elements of E
$\mathbf{P}(E)$	power set of E
$\langle a_1, a_2,\ldots, a_n\rangle$	an ordered sequence of n elements or n-tuple
$f: A \rightarrow B$	function which maps each member of A to a member of B
$f(x)$	the member of B to which f maps x, where $x \in A$
$\wedge$	logical conjunction, "and"
$\vee$	logical disjunction, "or"

List of Abbreviations

CB	Connection Block
DB	Diagnostic Block
ECB	Exception Control Block
ELRG	Event Log Report Generator
HIP	Host Interface Program
LDD	Link Device Driver
MD	Mapping Description
MIMD	Multiple Instruction, Multiple Data streams
MPAXS	Multiple Processor Asynchronous eXpandable System
OS	Operating System
PCB	Process Control Block
PID	Process ID, also pid
PTD	Process Type Description
PND	Physical Network Description
SCB	Station Control Block
SG	Script Generator
SND	Software Network Description
TCB	Timer Control Block
VLN	Virtual Link Number

Trademarks

UNIX is a trademark of AT&T.

Occam is a trademark of INMOS Limited (a member of the SGS-THOMSON Microelectronics Group).

NeXT is a trademark of NeXT Computer, Inc.

CrossCode is a trademark of SOFTWARE DEVELOPMENT SYSTEMS, INC.

Sun and Sun-3 are trademarks of Sun Microsystems Incorporated.

Ethernet is a trademark of Xerox Corporation.

Chapter 1 Introduction

1.1 Concurrent Software

This work describes the development of a tool to explore problems in the field of concurrent software. Concurrent software uses programming techniques which explicitly represent potential parallelism in programs, typically through the use of multiple processes which execute concurrently. Important constructs over and above those found in ordinary sequential computer languages deal with the communication and synchronization needs of concurrent processes. Concurrent programming techniques are particularly suitable for embedded systems, which must deal with a large number of sensors and actuators at the same time.

Concurrent execution of processes may be implemented by interleaving the execution of processes on a single processor, or by executing them in parallel on a number of processors in a network. In the latter case, the processes are said to have been "distributed" over the network. Distribution of a program allows exploitation of parallelism to improve time performance of the program. This extends the applicability of concurrent programming techniques to include any computationally intensive software (such as scientific programming) for which a shorter execution time is desired. Distribution of processes may also be motivated by reasons other than speed-up, such as cost-effectiveness, reliability, or the desire to utilize special purpose processors in a system.

1.2 Embedded Systems

In this work, we are primarily interested in concurrent software for embedded systems. The remainder of this document should be considered within the context of embedded systems. An appropriate environment for mapping concurrent software to parallel hardware is developed. A concurrent program is formulated as a software (virtual) network which is mapped to a physical network. NetKernel facilitates exploring the interplay between virtual and physical topology in determining performance. Different virtual and physical topologies are easily realized and tested.

An embedded system performs a predefined set of tasks upon dedicated hardware with minimal operator intervention. It responds to and generates many asynchronous signals and commands. Its performance may be improved by distributing activities to different processors. Since dedicated hardware is used, the topology of the hardware may be specially arranged to provide good performance for the software network of a particular embedded system. The processors are equipped with a minimal set of special-purpose peripherals, often lacking general-purpose facilities such as memory management hardware and directly accessible file systems. Special-purpose processors are often included as key components of an embedded system.

The allocation of software elements to physical resources is referred to as mapping. For an embedded system, it may not be obvious how to arrange the hardware (topology) and map the software so that performance and economic objectives are met. NetKernel does not directly address this problem, but provides a platform upon which it may be explored. [Fernandez89] states the *module allocation problem* where the goal is to minimize total execution time. (Other goals expressed in terms such as response time or throughput are also used for embedded systems.) Both processing time and interprocessor communication time enter into the minimization. The use of parallel processors and parallel links provides a means of reducing both times, respectively. Furthermore, irregular requirements for processing power and communication bandwidth in a software network make the use of heterogeneous hardware attractive for both the processors and the links. As an example, an embedded system might contain specialized image processors in a fast network combined with general-purpose processors in a slower network. The current implementation of NetKernel supports parallel M68000 processors and parallel serial links. The design allows for other types of processors and links to be added so that heterogeneous systems can be explored in the future.

1.3 Motivation

NetKernel is an initial effort in the MPAXS (Multiple Processor Asynchronous eXpandable System) project at the Department of Electrical Engineering, University of Alberta. The purpose of the MPAXS project is to develop hardware and software tools for the exploration of concurrent software on parallel hardware. This includes the areas of programming, mapping, and performance modelling. NetKernel is conceived as a bottom layer of distributed software to facilitate this exploration.

NetKernel's role on a parallel system is analogous to that of other kernels found on single processor systems. NetKernel will provide basic support such as process scheduling, interprocess communication, and input/output. Similar to other kernels, higher level programming paradigms may be built on top of NetKernel. A general programming paradigm will be selected for NetKernel on the basis of three aspects of concurrent software to be explored:

- **A1** mapping of concurrent software to heterogeneous parallel hardware,
- **A2** performance modelling and prediction,
- **A3** net-oriented programming paradigms.

Items A1 and A2 are currently active research areas. It is postulated that net-oriented programming paradigms (A3) provide a useful approach. The reasoning behind this approach will be presented following some discussion of parallel architectures and programming paradigms. This discussion serves to place the NetKernel paradigm within current classifications, and also indicates parallel architectures suitable for execution of NetKernel.

1.3.1 Parallel Programming Paradigms and Architectures

Ambler [Ambler92] defines a programming paradigm as:

> A programming paradigm is a collection of conceptual patterns that together mold the design process and ultimately determine a program's structure. Such conceptual patterns structure thought in that they determine the form of valid programs. They control how we think about and formulate solutions, and even whether we arrive at solutions at all.

Duncan [Duncan90] defines a parallel architecture as:

> ...a parallel architecture provides an explicit, high-level framework for the development of parallel programming solutions by providing multiple processors, whether simple or complex, that cooperate to solve problems through concurrent execution.

Ambler describes two general classes of paradigms which explicitly represent concurrency: synchronous paradigms and asynchronous paradigms. In synchronous paradigms, identical operations are performed in parallel on many data elements. In asynchronous paradigms, multiple processes execute in parallel, communicating and synchronizing with one another to accomplish some task. The operations performed by different processes need not be identical.

These two classes of programming paradigms are closely related to the two major divisions of parallel computer architectures identified by Duncan: synchronous and multiple instruction multiple data[1] (MIMD). Synchronous architectures execute related parallel operations on multiple processors in lockstep enforced by central control. Synchronous architectures are appropriate for synchronous programming paradigms, as well as related paradigms involving central control, such as systolic and vector processing. MIMD architectures execute independent parallel operations on multiple processors which are primarily autonomous. This type of architecture is well suited to executing multiple processes asynchronously on different processors. Duncan also identifies a third category of architectures, MIMD-based paradigm, which are asynchronous with multiple instruction and data streams, but also have "a distinctive organizing principle as fundamental to its overall design as MIMD characteristics" [Duncan90]. Examples include dataflow, reduction, and wavefront architectures.

---

[1] The term MIMD originates in Flynn's classification, [Flynn72].

The process oriented view of concurrent software falls under the asynchronous paradigm of programming. We have selected to explore this paradigm on a MIMD architecture. The choice of architecture was influenced by considerations of expandability, physical distribution, ease of custom configuration, and the possibility of building a machine of this architecture from readily available stand-alone computers.

Central control is considered to limit the performance of synchronous architectures as they are scaled up to more processors. Global control signals are particularly unacceptable for many embedded systems, where physical processors may be distributed throughout a factory. Another property of MIMD useful for embedded systems is that it can allow many different topologies, with the possibility of selecting a topology to optimize the performance of a particular program. A customized topology can improve performance for most parallel software, but is particularly useful for embedded systems because they tend to 1) be dedicated to one application over their lifetime, and 2) have irregular communication patterns.

An additional concern arises when considering performance gains from parallel processing: the amount of inherent parallelism available to be exploited in a programming problem. Some problems are inherently parallel (eg. image filtering), whereas others are inherently sequential (eg. computation of some nonlinear recurrence relations, such as the millionth iterate of $z_{n+1}=z_n^2+c$). *Identifying* the inherent parallelism in a problem is an important related topic which is not treated in this work. This thesis focuses on providing tools to *express* parallelism. [Courtois85] provides a general discussion on the decomposition of complex systems. Some criteria used to identify concurrent activities for embedded systems are presented in [Gomaa84]. For a text on parallel algorithms, the reader is referred to [Quinn87].

1.3.2 Message Passing vs. Shared Data

The asynchronous programming paradigm has been selected for NetKernel. There are two major refinements of the asynchronous paradigm: message passing and shared data. These are associated with the two major types of MIMD architecture, distributed memory and shared memory, respectively. The message passing and shared memory paradigms differ in the mechanisms used for interprocess communication and synchronization. However, they are equally powerful in that they can solve the same problems. We shall now consider these mechanisms and select one of them for NetKernel.

In the message passing paradigm, processes communicate and synchronize by sending and receiving messages. This is naturally implemented on distributed memory architectures, where processors are interconnected by many communications links. In the shared data paradigm, processes communicate and synchronize through shared data structures in a common address space. This is naturally implemented on shared memory architectures, where all processors have access to a common memory.

Shared memory generally provides higher performance for small numbers of processors, but becomes a bottleneck when large numbers of processors are used. Communication links between processors operate independently, which results in better expandability because the total available communications bandwidth can be increased by adding more links along with additional processors. Efforts to combine the performance of shared memory with the scalability of communication links have resulted in mixed architectures. These contain clusters (of processors) which are interconnected by links. Each cluster of processors has its own shared memory.

The choice of paradigm is no longer clear for mixed architectures. It is undesirable to employ both mechanisms, using each for processors connected in the corresponding fashion. This makes programs unnecessarily dependent upon the hardware configuration. Rather, a single mechanism should be uniformly available throughout all processors. From a system design perspective, it is easier to achieve message passing through shared memory than the other way around[1]. Since communication links present an undeniably message-oriented interface, message passing is almost always used in a lower layer.

In the interests of generality, NetKernel supports architectures which mix shared memory and links. The message passing paradigm will be supported uniformly throughout all processors. Message passing was selected because it is more easily realized on mixed architectures than shared data. This choice is consistent with the kernel's role of providing a low-level paradigm on top of which other paradigms may be realized.

1.3.3 Net-Oriented Programming Paradigms

The term "net-oriented" has been adopted to describe the programming paradigm developed in this work. This is a refinement of the message passing, asynchronous paradigm, which has been described above.

Network representations of computer software are commonly used in software design and modelling. Examples include data flow diagrams, petri nets, and queueing networks. Some asynchronous programming paradigms allow networks of processes to be established. These networks of processes are conceptually similar to the network representations used for design and modelling. The nodes used in modelling and design notations typically represent transformations or functions which can be realized by processes of the asynchronous paradigm. The arcs typically represent information flow which can be realized by message passing. Hence the concepts expressed in network models and design notations are easily realized in programs with the message passing, asynchronous paradigm.

---

[1] Message passing is easily achieved through shared memory by the use of data structures such as buffers, queues, mailboxes, and semaphores. Shared data, on the other hand, is more difficult to achieve through communication links, since it must be copied between memories by sending messages through the links. The coherence of the data needs to be maintained in the presence of concurrent accesses, while still providing adequate performance.

It would seem useful for a programming paradigm to employ exactly the same abstract elements as are being used for design and modelling. Such elements might include the queues of queueing networks, or the transitions of Petri nets. The use of the same elements for different aspects of software development should improve the consistency and reduce the translation effort between phases.

A net-oriented paradigm will now be defined as a message passing asynchronous paradigm in which a program is structured as a network of processes. Interaction between processes is determined by the connections in the network. These connections indicate the flow of messages for purposes of communication and synchronization. High-level net-oriented paradigms may support specific features of design or modelling notations, such as the queues of queueing networks or transitions of Petri nets.

A benefit of the message passing asynchronous paradigm, which carries over to the net-oriented refinement, concerns mapping of software to parallel hardware. Processes form convenient units of distribution. Once a program has been expressed using the paradigm, it can be allocated to parallel hardware by specifying the processor on which each process should execute. Such an approach facilitates experimental evaluation of different mappings to compare performance. This should prove particularly useful in the effort to develop tools which predict performance and select good mappings. An even more ambitious tool would identify the best arrangement and interconnection of available hardware for a particular program.

A final aspect of net-oriented paradigms which will be mentioned here concerns the potential for manipulating the net of processes as it executes. This may be particularly useful for embedded systems, for which it is desirable to make on-line changes to software. One or more replacement processes may be loaded at runtime. By changing the connections between processes "on-the-fly", the replacement processes can be integrated into an executing network of processes. Of course there may be additional concerns with respect to the coherency of such an operation, such as transferring data structures of replaced processes.

The following section introduces general features of NetKernel and its associated environment. These features have been selected to support the concept of a net-oriented programming paradigm, as set forth above.

1.4 NetKernel

NetKernel is an experimental kernel for executing concurrent software on heterogeneous distributed systems. It provides low-level support for net-oriented programming paradigms, including paradigms not traditionally used for computer programming, such as queueing networks and Petri nets. Programming environments for such paradigms may be implemented on top of NetKernel. In particular, it is anticipated that NetKernel will be used to explore programs compiled from an intermediate design notation based on Petri Nets. Netkernel and its associated development environment provide the basic facilities required for process management, interprocess communication, and configuration.

A NetKernel program is a software network of processes which is mapped to a physical network of processors. All processes execute concurrently, with the potential to exploit parallelism by mapping processes to different processors. Message passing is the only form of interprocess communication directly supported by NetKernel. Other forms of interprocess communication, such as shared data to enhance performance, are possible by introducing mapping constraints upon processes.

The runtime kernel and development environment for NetKernel are described. These have been implemented on the MPAXS hardware. The design and interface of the kernel are described. NetKernel takes the "microkernel" approach which has recently gained momentum for operating systems software. The development environment for NetKernel programs includes tools to facilitate mapping, loading, start-up, and controlling software networks.

1.5 History of the MPAXS Project

The MPAXS project was conceived in 1989 approximately one year before this work began. Its goal is to provide an experimental parallel computer system to explore issues in the mapping and programming of concurrent software. Computers being used for an undergraduate lab course at the department were selected as the initial building blocks for MPAXS. These provide an economical source of simple units to be interconnected.

Prior to this work, undergraduate projects on providing shared memory hardware and software evaluation were underway. The shared memory project is still an ongoing undergraduate project. When shared memory is ready, it can be integrated with the work described in this thesis.

1.6 Outline

The organization of this thesis reflects the development process of NetKernel. Chapter 2 presents the requirements and initial description. Chapter 3 presents the high level design, followed by a refinement of the design, followed by some implementation information. The focus of this thesis is NetKernel itself, but Chapter 4 does introduce some of the development tools associated with NetKernel. The appendices present further details on implementation, including some example programs.

Chapter 2 presents the programming paradigm and architecture of NetKernel. A program is formulated as a collection of processes arranged in a *software network*. The architecture introduces two additional key concepts for NetKernel: *physical network* and *mapping*.

Chapter 3 presents the design and implementation of NetKernel which we have realized on M68000 targets interconnected by serial links. The design consists of 1) a microkernel which supports process switching and interprocess communication, and 2) a collection of kernel processes which provide additional services. Sections Section 3.5 and Section 3.6 go into some depth on the design. These might be skipped by a reader who is less interested in details.

Chapter 4 describes the tools which have been set up to facilitate development and execution of NetKernel programs.

Chapter 5 provides an overview of environments and paradigms for process oriented concurrent programming. Some existing parallel systems similar to NetKernel in terms of goals or approach are described and compared to NetKernel.

Chapter 6 presents our conclusions and ideas for future research. NetKernel is intended as an environment to support future work, especially investigation of a design notation based on Petri Nets which is being proposed at our lab.

The appendices present some aspects of the NetKernel interface and sample programs. These are intended to further illustrate our system and demonstrate its capabilities. In particular, the ability to flexibly map concurrent software to parallel hardware is demonstrated.

Chapter 2 NetKernel Paradigm and Architecture

2.1 Overview

The NetKernel programming paradigm and the general target architecture are described. The paradigm is a net-oriented, as introduced in the previous chapter. Its major feature is that programs are expressed as a software network of asynchronous processes. Processes have local input queues and outlets for message passing. Processes are mapped to a physical network of stations (i.e. processors which run NetKernel). Stations may be interconnected by point-to-point links, such as serial channels, and shared access links, such as Local Area Networks (LANs) and shared memory.

The NetKernel software runs on each station in the physical network. Each station's kernel manages the execution of asynchronous processes at that station. Message passing primitives are available to all processes. The kernel provides transparent support for message passing between processes at different stations.

The development tools run on workstations, providing runtime control of processes throughout a physical network. The Host Interface Program (HIP) is the primary tool. NetKernel is designed to facilitate runtime control by higher level software. This enables HIP to provide an interface for loading, execution, and control of processes. It also supports establishment and breaking of connections between processes, which is performed at runtime to arrange processes into software networks.

This chapter provides a high-level description which is applies to a wide range of hardware (i.e. processors and links). NetKernel is intended to run on platforms ranging from simple microprocessors without virtual memory, to workstations with native operating systems. Naturally, there will be some station specific details (eg. process scheduling). These are discussed for our implementation in the following chapter. The presentation of this chapter is divided into two parts. The first part presents the NetKernel programming paradigm, independent of any architectural considerations. The second part presents the general architecture and framework upon which we support the paradigm.

2.2 Operational Requirements

The purpose of the NetKernel paradigm is to provide an environment for the realization of higher level net-oriented programming paradigms on parallel hardware. Two major requirements have been identified for NetKernel:

B1 interprocess communication and synchronization,
B2 flexible mapping of concurrent software to parallel hardware.

Interprocess communication and synchronization is a basic requirement for any multiprocessing system. General mechanisms suitable for higher level net-oriented paradigms on parallel hardware should be provided.

Flexible mapping of concurrent software to parallel hardware requires that the allocation of a concurrent program to physical hardware is described independently of the program itself. It should be possible to experiment with different allocations of a concurrent program to one or more physical networks without changing the program itself.

Part A: Paradigm

The NetKernel paradigm defines the structural organization of concurrent programs as a software network and primitives for message passing between processes. It is not associated with any particular programming language. It may be implemented directly as a computer language, or added to an existing language using library functions. It is possible that processes of a software network could be written in different languages. In general, however, it is desirable for a single language and interface to be available across different station types. This allows for a process type to be mapped to different types of stations, with only one source program needed for it.

2.3 Software Network

A NetKernel program executes as a collection of asynchronous processes arranged in a software network. The terms software network and NetKernel program are used interchangeably, and should not be confused with sequential programs. The description of a software network is separated into two levels: 1) the description of individual processes, and 2) the description of how processes are instantiated and connected together. Benefits of this separation will be discussed in Section 2.7.

2.3.1 Process

A process is a sequential program in execution. Among the instructions a processes may execute are system calls to NetKernel. This allows a process to access services provided by the kernel, including message passing. Conceptually, a process may be illustrated as shown in Figure 1. A set of input queues and a set of outlets are associated with each process.

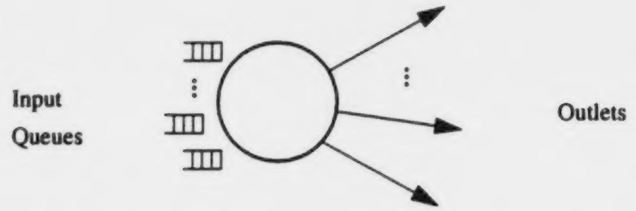


Figure 1 Process

Messages from other processes are received in input queues. Messages are sent through outlets. When a process sends a message through one of its outlets, that message will be deposited into the input queue to which the outlet has been connected. Connections between outlets and input queues are usually established by higher level software, although it is possible for processes to establish their own connections. Processes may also reply to messages which they have received, which does not involve the use of an outlet. Processes synchronize with one another by waiting for the arrival of messages into one or more input queues. The message system is described further in Section 2.4, "Message System".

More formally, a process is a triple $p=\langle p_S, p_Q, p_O\rangle$ with

- p_S = sequential program,
- p_Q = set of local input queues,
- p_O = set of local outlets where $p_Q \cap p_O = \emptyset$.

A process is considered to be of a specific *type*, where the type determines the sequential program, input queues, and outlets for the process. Two processes x and y of the same type are identical if their sequential programs are the same and they have the same number of input queues and outlets, i.e. $x_S = y_S$, $|x_Q| = |y_Q|$, $|x_O| = |y_O|$.

2.3.2 Software Network

A software network is a collection of connected processes. Outlets are connected to input queues. Connections between processes are established through commands to NetKernel. Presently there is no kernel support for broadcast or multicast, so an outlet may be connected to at most one input queue. Multiple outlets, however, may be connected to the same input queue. Figure 2 illustrates an example software network.

A software network is formally defined as $s=\langle P, C\rangle$.

- P = set of processes where
  $(\forall x,y \in P)\ ((x \neq y) \rightarrow (x_Q \cap y_Q = \emptyset))$ no processes share input queues, and

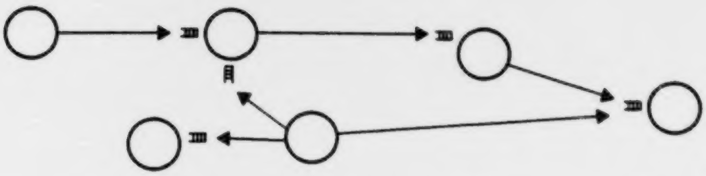


Figure 2 Example of a software network

$(\forall x,y \in P)\ ((x \neq y) \rightarrow (x_O \cap y_O = \varnothing))$ no processes share outlets.

The function *c* is introduced here in order to define C below.

- $c : O \rightarrow Q \cup \varnothing$ is the interconnection function of S, which maps each outlet of the software network to an input queue or $\varnothing$.
  $O = \{\ x \mid x \in p_O,\ p \in P\ \}$ is the set of all outlets in the software network.
  $Q = \{\ x \mid x \in p_Q,\ p \in P\ \}$ is the set of all input queues in the software network.

The function *c* defines the interprocess connections in a software network. For a particular outlet *o*, $c(o)=q$ means that outlet *o* is connected to input queue *q*. If $c(o)=\varnothing$, then outlet *o* is not connected to any input queue (i.e. outlet *o* is unused).

- $C=\{\ \langle o,q \rangle \mid o \in O,\ c(o)=q \neq \varnothing\ \}$ = set of connections.

An outlet joined to an input queue is referred to as a *connection*[1]. The set of connections C contains a member <o,q> for every connected outlet in the software network.

2.3.3 Provision of Services by Kernel Processes

NetKernel provides various services such as I/O, process management, outlet connection, object code loading, and timers. These services are implemented by *kernel processes* which begin executing automatically when the kernel is started. This approach has recently been adopted for some operating systems, which consist of a small core and provide most services through processes which are implemented similarly to user processes.

NetKernel's services are accessed by sending messages to kernel processes. This makes the services easily accessible to user processes through message passing. The services are also accessible from external I/O devices, such as terminals, through a command line interface to the message system.

Software networks will typically contain both user and kernel processes. As an example, consider the hypothetical software network in Figure 3. Kernel processes are lightly shaded to distinguish them from user processes. Process *ts* is a kernel process providing a timer service. Process *mon* is a user process. The connection from *mon* to *ts* allows *mon* to send messages (eg. request/disable a timer) to *ts* through its outlet. The kernel process responds by sending reply messages back. Process *mon* accepts reply messages from *ts* into its input queue *rep*. There is no outlet from *ts* to *mon* since *ts* replies to messages from *mon*, rather sending messages to *mon* through an outlet.

---

[1] As used here, the term connection does not have the same implication as in telecommunications, where it implies that communication resources or bandwidth have been reserved between a sender and receiver.

2.3.4 Input/Output

Processes referred to as device drivers perform direct I/O with devices at a station. The kernel processes will typically include one or more device drivers. Other processes, including user processes, may request I/O operations by sending messages to device drivers. It is also possible for users to load their own device drivers.

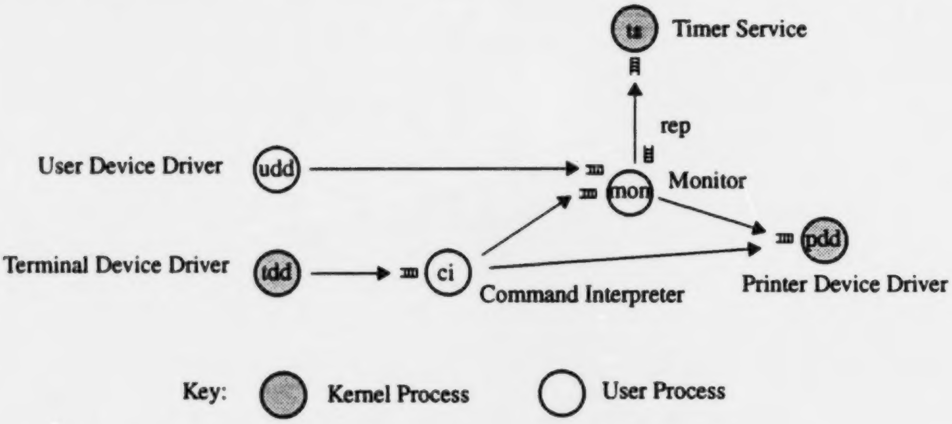


Figure 3 Hypothetical software network to monitor a reactor

As an example, consider the following hypothetical situation. Software is to detect events from a reactor. Events should be logged to a printer. When certain events fail to occur within a timeout period, this is also logged to the printer. An operator enters commands at a terminal. These commands select which events should be monitored, and may specify lengths for various timeout intervals. The operator may also send messages directly to the printer, in order to add comments to the log.

A software network for this problem is shown in Figure 3. A user-supplied device driver, *udd*, detects events and sends corresponding messages to Monitor. Monitor sends messages to Timer Service to manage its timers. When a timeout or an event which is being monitored occurs, Monitor sends a log message to the Printer Device Driver. For each command line typed at the terminal, Terminal Device Driver sends the line to Command Interpreter. Command Interpreter processes command lines. For each valid command,

Command Interpreter sends messages to 1) Monitor for selecting events or timeout length, or 2) Printer Device Driver for printing comments.

This example illustrates how user processes interact with kernel processes to make use of kernel services and perform I/O. A user supplied device driver is also shown. This latter feature is particularly useful for embedded systems development.

2.3.5 General Process Models

This subsection describes some general relationships between interacting processes. These relationships will be evident among the kernel processes which are described in the next chapter. They are also useful for user processes.

Most processes are similar in that they respond to events. There are two types of events: the arrival of a message and the arrival of an exception. Mechanisms for handling exceptions are not discussed in this chapter because they are dependent on the station type. Processes may generate events for other processes by sending messages.

The code of a process which responds to events can be structured in a simple fashion. The process begins by initializing itself and then waits for an event. The occurrence of an event triggers response activities by the process. It responds by performing various operations such as updating data structures, sending and receiving messages, accessing peripheral devices, and updating its internal state. Upon completion of the response, the process then goes back to waiting for the next event.

Based on its response, a process is considered to be either a server or a transform process. These correspond to the client-server and data flow models, respectively. NetKernel supports and makes use of both models. These concepts also apply to user software networks. The reactor example (Figure 3) contains both server and transform processes.

2.3.5.1 Client-Server

Many processes provide services which are accessed using the client-server model. In this model, a *client* process sends requests to a *server* process. Upon receiving a request, the server performs some action and sends an answer or confirmation back to the client. A single process may act as both server and client in different contexts. This is because a process may both provide a service and make use of other services.

Figure 4 illustrates client-server relationships between processes. Figure 4a shows a single client and server. The server has an input queue into which requests are accepted. A client sends requests through an outlet connected to that input queue. When sending a request, the client specifies an input queue of its own into which replies from the server should be placed. The graphical notation for software networks has been extended with a hollow arrowhead which points to the input queue selected by a client for replies to messages it sends through an outlet. Figure 4b shows a more complex software network with a process which behaves as both client and server. It also demonstrates how two clients, *X* and *Y*,

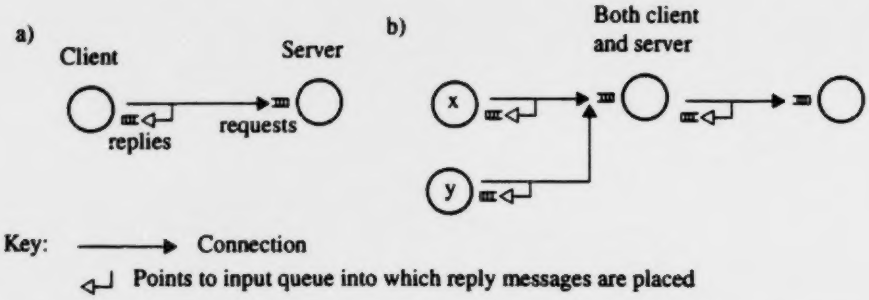


Figure 4 Examples of client-server relationship between processes

may make use of the same server. These clients could be from otherwise unrelated software networks, totally unaware of each other's existence. In the earlier reactor example, Timer Service is a server.

Client-server models often make use of a name service or directory. Clients retrieve a port or address for a server from the directory before communicating with that server (eg. the name server in Mach, [Loepere90, pp. 77-84]). Currently, no such mechanism is provided for NetKernel, although such an extension is conceivable. The current strategy based on connections is simpler for software networks with static patterns of communication. The connections are usually established by a development tool when a software network is loaded, rather than by the clients themselves.

2.3.5.2 Data Flow

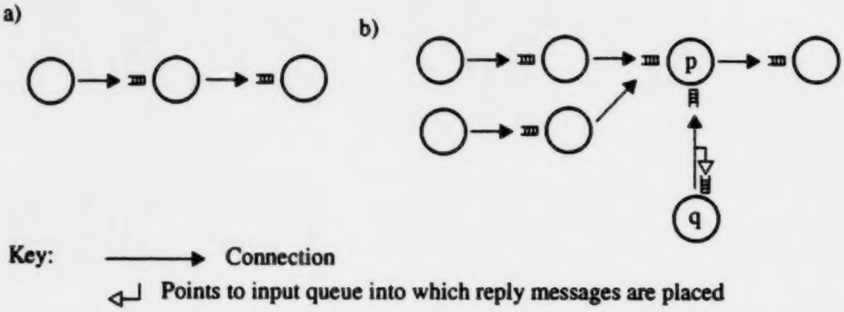


Figure 5 Examples of data flow configuration of processes
a) Simple data flow b) Mixed data flow and client-server

In this model, data flows from one process to the next, being transformed by each process. Hence, processes in this model are referred to as *transform* processes. Processes may also perform input and output with the environment. This model differs from the client-server model in that reply messages are not used.

In NetKernel, data flowing between processes takes the form of messages. Figure 5 illustrates a data flow relationship between processes. Figure 5a shows a simple arrangement of three processes to form a pipeline. Data flow networks are characterized by one way streams which may split or merge. Figure 5b shows a more complex software network with both data flow and client-server behavior. Process *p* transforms a stream flowing from left to right. It also provides a service to *q*. As a simple example, the service might return a count of the number of messages which have flowed through *p* along the left to right stream. In the earlier reactor example of Figure 3, most of the processes are transform processes.

2.4 Message System

Message passing has been selected as the paradigm for interprocess communication (Section 1.3.2). This section motivates and describes message passing in the NetKernel paradigm.

2.4.1 General

The two basic requirement for the message system are configurability and transparency. Configurability allows processes to be used as building blocks from which software networks are constructed. It should be possible to reuse the sequential program of a process in two different software networks without modifying the code. Transparency requires that there is no difference, from the perspective of two processes exchanging a message, whether they are mapped to the same station, or to different stations connected by links and intermediate stations. These two requirements resulted in system calls for which the calling process only refers to objects which are local to itself (i.e. the parameters of a system call may refer to input queues or outlets of the caller, and messages previously received by the caller, but not to input queues or outlets of other processes).

The NetKernel paradigm defines a general model and system calls for accessing an underlying communication service. However, NetKernel does not fully specify the characteristics of the underlying service. Ideally, it is a reliable, ordered datagram service. Real implementations, however, will often provide fewer guarantees.

2.4.2 Primitives

Five primitives associated with the message system are introduced below. Only the basic functionality of each primitive is presented. An implementation may provide these as constructs of a programming language, or as calls added to a language. Information on our implementation is presented in the next chapter (Section 3.5.5).

EXAMINE() → ReadyQueueSet

Return value indicates which of the caller's input queues are *ready*. A queue is considered ready if it contains at least one message.

REPLY(OldMessage, Message)

Send a reply to a previously received message. If the sender of the previously received message had indicated that replies were not to be accepted, then the call fails.

SEND(Outlet, Message)

Send a message through one of the caller's outlets. If the specified outlet has been connected to an input queue, then the message will be deposited into that queue. Otherwise, the call fails.

Note on acceptance of reply messages:

SEND and REPLY are the only calls which send a message. In both cases, the sender specifies whether replies to the message will be accepted. If replies will be accepted, then the sender also specifies into which of its input queues any reply messages are to be placed. (For clarity, the parameters used for this are not shown above.)

TAKE(QueueSet) ⟶ Message

Removes and returns one message from the subset of the caller's input queues specified in *QueueSet*. If none of the specified queues contain any messages, then the call fails.

WAIT(Condition) ⟶ ReadyQueueSet

Wait until a condition on the caller's input queues is true. If true, the call returns immediately. Otherwise the caller is blocked until the condition becomes true. Return value indicates which of the caller's input queues are ready. A queue is considered *ready* if it contains at least one message. *Condition* is a boolean expression made up of terms M_q, where

$M_q \equiv$ Input queue q is ready, $q \in p_Q$, where p=calling process

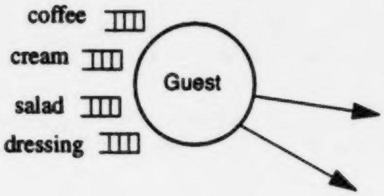


Figure 6 Guest process

For example, consider the *Guest* process in Figure 6. *Guest* waits to be served either coffee with cream, or salad with dressing. By performing the following wait call, *Guest* will be blocked until either queue *coffee* and queue *cream* are both ready, or queue *salad* and queue *dressing* are both ready.

wait($(M_{coffee} \wedge M_{cream}) \vee (M_{salad} \wedge M_{dressing})$)

NetKernel limits *Condition* to two-level boolean expressions making use of conjunction and disjunction, but not negation. That is, *Condition* takes one of the following two forms:

$(M_{a1} \wedge \ldots \wedge M_{an}) \vee (M_{b1} \wedge M_{bn}) \vee \ldots \vee (M_{x1} \wedge M_{xn})$, or
$(M_{a1} \vee \ldots \vee M_{an}) \vee (M_{b1} \wedge M_{bn}) \vee \ldots \vee (M_{x1} \wedge M_{xn})$

2.4.3 Messages

NetKernel does not define the structure, semantics, or typing of messages. There are two basic requirements for memory management:

C1 Arrived messages appear in the address space of the receiving process,

C2 The memory occupied by a message can be reused when the message is no longer needed.

These requirements may be realized at the language level by declaring messages as variables. Or they may be realized in a less structured fashion using system calls for the dynamic allocation and freeing of memory blocks.

2.4.4 Process Blocking and Timeouts

A process whose execution is delayed while it waits for one or more events to occur is said to be blocked. A blocking call is one which may cause the calling process to become blocked. An alternative to blocking is polling. A process can simply run in a loop, checking over and over again to see if an event has occurred. Polling is undesirable because it wastes processor time. It is appropriate to consider the use of timeouts with blocking calls. Timeouts provide a means for processes to take alternative actions when an event does not occur within a certain interval of time.

The only NetKernel primitive which needs to block is WAIT. As described earlier, WAIT causes the calling process to wait until some condition upon its input queues is true. The SEND and REPLY primitives may also be blocking, depending on the underlying message service. If the underlying service incorporates flow control, then the sender may be blocked until the receiving process has an empty queue slot for the message being sent. If the underlying service incorporates congestion control, then the sender may be blocked until the network can accept the message.

Processes using these primitives may need to perform other activities with specific timing requirements. Hence, timeout options should be provided for system calls which block (i.e. WAIT, and possibly SEND and REPLY). This avoids blocking a process for unpredictably long periods of time.

There is a natural approach to providing timeouts for the WAIT call. It makes use of the kernel process which provides a timer service. Consider a process *M* which wishes to wait with timeout. First *M* establishes a timer with the required timeout period. When the timeout occurs, the timer service will send a timeout message to an input queue of *M*. Process *M* then includes the arrival of a message into that queue as one of the conditions for the WAIT call. Should one of the other conditions be satisfied first, *M* will disable the timer upon returning from WAIT. This is the approach used by the Monitor process in the reactor example of Section 2.3.4.

If SEND or REPLY are blocking, then they may also require a timeout mechanism in real-time applications. Extensive use of timeouts in a program, however, can make it hard to understand. For this reason, it may be preferable to use non-blocking versions of SEND and REPLY for real-time parts of a system.[1]

2.4.5 Transparency and Configurability

Processes are building blocks from which software networks are built. To facilitate the arrangement of processes into networks, it is important that incorporating a process into a software network does not require making any changes to the sequential program of the process. This is especially critical for NetKernel since connections between processes are established by the runtime system. Hence the code of a process should not contain any direct references to other processes.

All of the message system calls satisfy this requirement. SEND specifies its destination by referring to a local outlet of the caller. The REPLY primitive specifies its destination by referring to a previously received message. EXAMINE, TAKE, and WAIT all refer to local input queues of the caller. The runtime system maintains a database of connections between input queues and outlets, which can be dynamically modified. This provides the required configurability.

This approach also provides transparency. The runtime system allows connections to be made between processes on the same or different stations. From the perspective of a process, there is no distinction between the two cases. A process simply sends a message through an outlet and the runtime system is responsible for delivering the message to the connected process. Similarly for replies, there is no distinction between replies to messages of local and nonlocal origin.

---

[1] Our implementation will be revealed to use non-blocking versions with the assumption that the underlying hardware and message service is capable of supporting the real-time demand placed upon it.

Part B: Architecture

The NetKernel architecture is general in that it does not specify any particular type of hardware or organization. Instead, it classifies the types of hardware which may be used and defines the mapping of software networks to physical networks of hardware.

2.5 Physical Network

A specific configuration of parallel hardware is referred to as a physical network. A physical network is built from stations, links, and I/O devices. Links are used to interconnect stations. A station may also have one or more attached I/O devices.

2.5.1 Stations

Processor Class →

Kernel Support ↓

	Host	Target
NetKernel (Station)	Functions -NetKernel software execution -Development environment	Functions -NetKernel software execution
Non-NetKernel (I/O Device)	Functions -Foreign software execution -Development environment	Functions -Foreign software execution

Figure 7 Processor classification and functions

A station is a processor which runs NetKernel. Stations running NetKernel provide an environment for the execution of software networks. It is possible to incorporate processors which do not run NetKernel into a physical network, treating them as I/O devices. The classification of processors in a physical network is illustrated in Figure 7.

Processor Class: Host vs. Target

We distinguish between two classes of processors: host and target. Host processors, typically workstations, support development tools and other utilities. The development environment may use the host's native operating system and programs for many services, such as file system and editing. Target processors provide raw processing power and real-time response. Targets can have simple hardware, lacking virtual memory or secondary storage.

Kernel Support: NetKernel vs. Non-NetKernel

We also distinguish between processors which run NetKernel and those which do not. A processor which runs NetKernel is referred to as a *station*. It is desirable for as many processors as possible in a network to run NetKernel, in order to support features such as transparent interprocess communications. However, there are reasons for not running NetKernel on a processor. Such processors may be included in the physical network as I/O devices. This is considered further under "I/O Devices" in Section 2.5.3.

A host may run NetKernel on top of its native operating system. In this case, NetKernel's performance on the host will not be as high as a direct implementation on the raw hardware. However, design and implementation of NetKernel on top of a multiprogrammed OS, such as Unix or Mach, can be far simpler, and the advantage of a uniform programming paradigm across targets and hosts is still achieved. A host may run the development tools independently of whether or not it runs NetKernel. Hosts which run NetKernel may also run user processes which are part of a NetKernel software network.

2.5.2 Links

Link Class →

Point-to-Point	Multiple access
RS 232 serial link Two ended parallel link	Common memory Ethernet Token Ring

Figure 8 Link classification and examples

A link is hardware which connects two or more processors to allow direct communication between them. It is useful to classify links as illustrated in Figure 8.

A point-to-point link connects exactly two stations. It is distinguished from multiple access links in that there is no need for the sender to explicitly select a receiver on the link. A multiple access link, on the other hand, connects two or more stations. The sender must explicitly select a receiver. The receiver is selected using some sort of addressing scheme, such as numbering all potential receivers starting from zero. Shared access links may also have the capability to broadcast to all receivers, although we do not presently utilize this capability.

2.5.3 I/O Devices

Many different types of I/O devices are possible. These include terminals, printers, sensors, actuators, etc. No attempt to classify these will be made. As was noted earlier, processors which do not run NetKernel may also be incorporated into the physical network as I/O devices.

The use of I/O devices in the physical network requires device driver processes (see Section 2.3.4, "Input/Output"). Each type of device has an associated device driver process, which must run at the station to which the device is attached. A single device driver process can handle multiple devices of the same type attached to a station. When devices are attached to different stations, however, then each station must run its own device driver.

2.5.3.1 Treating Processors as I/O Devices

It may be infeasible to run NetKernel or impractical to implement NetKernel on certain processors, for reasons such as the following:

- **D1** Effort of developing NetKernel for the processor is not justified,
- **D2** Overhead of executing NetKernel on the processor is unacceptable,
- **D3** Other system software is more appropriate for the processor.

Such processors may be utilized in conjunction with NetKernel by treating them as I/O devices. As an example, consider high performance image processor which acquires data, processes the data, and transmits the transformed data to a NetKernel station. Although NetKernel could probably be implemented on such a processor, it would be an unreasonable effort to implement a general purpose kernel for a processor which will be used in only one specialized way. Additionally, the overhead of executing NetKernel on the image processor would probably be unacceptable.

The use of specialized processors for performance reasons might seem contradictory when the intent is to provide a physical network which can be scaled to meet processing needs. However, specialized processors provide attractive solutions to specific aspects of a problem which need to be integrated into a larger solution. NetKernel accommodates integration and provides scalability where needed.

2.5.4 Physical Network

The physical network contains three basic types of hardware which have been described: stations, I/O devices, and links. Stations are interconnected by links, while I/O devices are considered to be directly attached to stations. An example of a physical network is shown in Figure 9.

This example is for a problem similar to the reactor example of Section 2.3.4, "Input/Output". Now there are three reactors to be monitored. Monitoring each reactor is a high speed digital signal processor (DSP) attached to its own target station. The DSPs do not run NetKernel, so they are considered to be I/O devices. Instead they run optimized signal processing algorithms and detect events from filtered signals. Each DSP sends information about detected events to its target station.

All three reactor target stations are interconnected by an Ethernet link. The development workstation is used to load and start the software network on the targets. An alternative to loading the software would be to place it into EPROM at all of the target stations.

A physical network p is formally defined as $p = \langle S, L_P, L_M, s, D, d \rangle$.

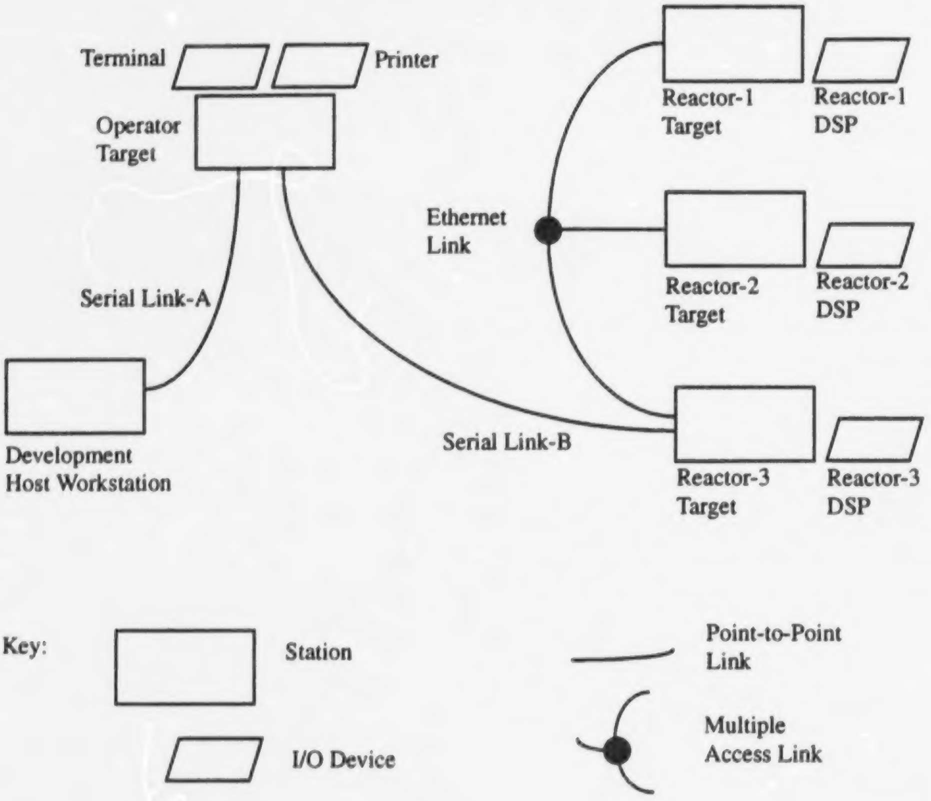


Figure 9 Hypothetical physical network

- S = set of stations.
- L_P = set of point-to-point links, L_M = set of multiple access links.
- $s : L \rightarrow \mathbb{P}(S)$ where $L = L_P \cup L_M$ = set of all links,
  such that $s(x)$ returns the set of stations directly connected by x, $x \in L$.
- D = set of I/O devices.
- $d : S \rightarrow \mathbb{P}(D)$ such that $d(y)$ returns the set of I/O devices attached to y, $y \in S$.

2.5.5 Variations in the Environment On Different Station Types

Since NetKernel is intended to run on stations ranging from simple embedded controllers to workstations, naturally there will be considerable variation of the environment in which processes execute. The NetKernel paradigm places as few restrictions as possible on acceptable environments. The minimum facilities for the runtime environment of a station are support for the elements of the paradigm and its primitives. These were presented earlier and will be summarized in Section 2.8.

NetKernel does not specify any scheduling or memory protection, for instance. Consider memory protection. On a workstation with an operating system, each NetKernel process could execute in its own address spaces. But on a simple embedded processor lacking memory management hardware, the processes could execute in a single shared address space. The ability to exploit specific features of particular stations (eg. scheduling provisions) is provided by the concept of secondary mapping, discussed below.

2.6 Mapping

A mapping is a description of the allocation of a software network to a physical network. It establishes which physical resources and underlying services are used to realize the abstract elements of the software network.

A NetKernel mapping is divided into two parts: primary and secondary. The primary part specifies fundamental allocations required by the NetKernel paradigm. The secondary part is purposefully left undefined. It describes mapping issues which are particular to an implementation. The related issues of mapping objectives and constraints will be considered briefly, although we do not yet provide any techniques for dealing with them.

2.6.1 Primary Mapping

A primary mapping specifies the following information:

E1 the allocation of processes to stations (for each process in the software network, this indicates the station on which it executes),

E2 the allocation of input queues to buffers (for each input queue in the software network, the length of the queue is specified).

A primary mapping for a software network $s=\langle P, C\rangle$ to a physical network $p=\langle S, L_P, L_M, s, D, d\rangle$ is defined as $M_1=\langle m_s, m_l\rangle$.

- $m_s : P \rightarrow S$ such that $m_s(p)$ returns the station on which process p executes, $p \in P$.
- $m_l : Q \rightarrow \mathbf{N}$ such that $m_l(q)$ returns the queue length allocated to input queue q, $q \in Q$ where $Q = \{\ x \mid x \in p_Q,\ p \in P\ \}$ is the set of all input queues in the software network.

2.6.2 Secondary Mapping

The secondary mapping is purposefully left undefined. Each implementation can define it based upon the intended application area and hardware supported. The secondary mapping establishes implementation specific details about the mapping of a software network to the elements of a physical network. These details will relate to the types of stations and services available in the physical network. The secondary mapping allows a software network to take advantage of features which may be unique to a particular physical network.

To elaborate, two possible areas for secondary mapping will be considered. The first is mapping of processes to scheduling priorities and policies. Scheduling schemes may be vary for different types of stations in a physical network. For instance, one type of station might support numeric priorities from 0-7. On another station type, several classes such as batch, interactive, and high priority might be supported. For every process, the secondary mapping would specify either a numeric priority or a class, depending upon the type of station to which the process is mapped.

Another possible area is the mapping of connections to different qualities or priorities of message delivery. Mapping choices would depend on the underlying message services available to the NetKernel runtime system. Different services distinguished by factors such as reliability, ordering, routing technique, performance, or flow control might be available.

2.6.3 Example

Figure 10 illustrates an example mapping. The software network of Figure 3 has been mapped to a physical network of two stations. The physical network is similar to that of Figure 9, except that only one reactor is considered and the host workstation is not included. This leaves only two stations, the operator target and reactor target. Each pro-

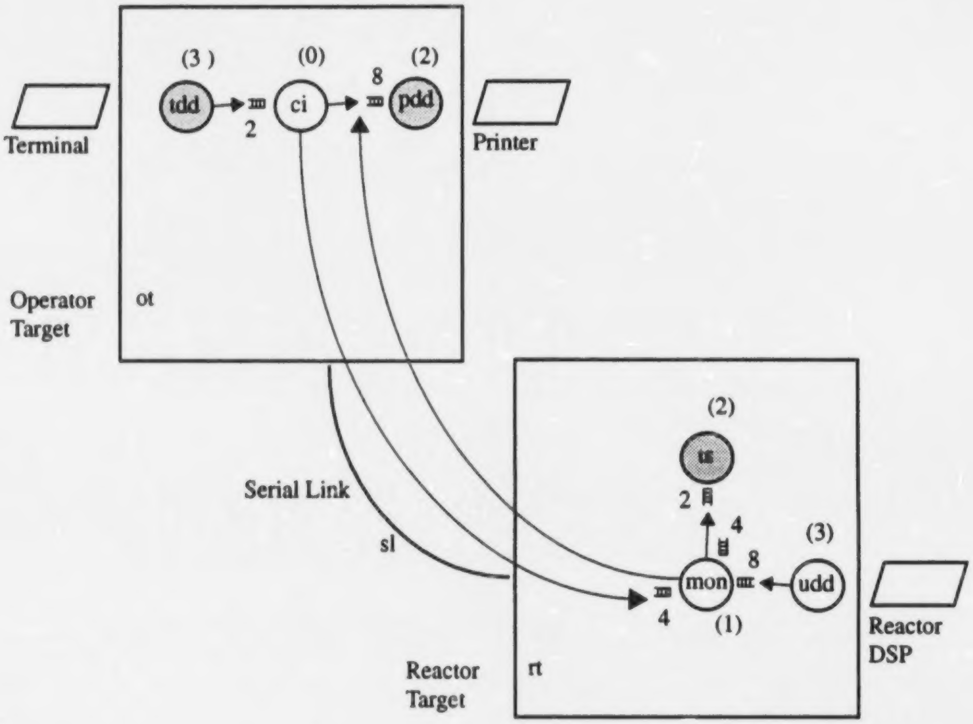


Figure 10 Example mapping

cess has been drawn inside of the station to which it is mapped. The length of each input queue is specified by a number at each queue. The process allocation and queue lengths constitute the primary mapping.

Figure 10 also shows other some other allocation details, which are part of the secondary mapping. For this hypothetical example, we assume that both stations support processes having execution priorities from 0 to 3. The bracketed number by each process indicates the priority at which the process runs.

2.6.4 Mapping Objectives

A software network is executed on a physical network to accomplish some task. How it is mapped to the physical network affects its performance. Mapping processes to different stations can improve performance through parallel execution. On the other hand, the resulting communication overhead between stations may diminish or even outweigh the benefits of parallel execution.

A designer has certain objectives to achieve. These goals provide a basis for comparing different mappings. One straightforward goal is to minimize execution time. More specific objectives are possible, such as particular throughput and response time requirements for activities performed by the software network. Response time goals are common for embedded systems. When there is flexibility in terms of the hardware to be used, minimizing its cost can be an important goal.

This work does not directly address the problem of how to select a mapping. By setting up a framework which facilitates mapping of software networks to physical networks, we hope to provide a basis for future research into that problem.

2.6.5 Constraints

The mapping of a software network to a physical network can be subject to constraints. Constraints reduce the number of possible mappings. Note that constraints differ from objectives. Constraints are direct restrictions on acceptable mappings, whereas objectives are goals to be achieved by finding a suitable mapping. Constraints on mapping are usually independent of each other, making it easier to determine whether or not they can be satisfied and to ensure their satisfaction where possible. Finding a suitable mapping to satisfy multiple objectives is much harder because each mapping decision usually affects several objectives.

One basic constraint which NetKernel imposes is that each device driver process must be mapped to the station to which its device is attached. So for the previous example, Terminal Device Driver and Printer Device Driver must both be mapped to the station Operator Target.

Many other constraints are possible. Three brief examples will be presented. A process could be constrained to execute on a particular type of station. This might occur because it is written in a language whose compiler is only available for that station type. Two or more processes could be constrained to execute on stations having access to a common memory. This might occur because the processes interact through shared data structures in a common address space. This is an example of using an interprocess communication mechanism other that of NetKernel's message passing, thereby introducing mapping constraints. As a final example, processes could be constrained not to execute on certain stations. This might occur because those stations do not have adequate resources (eg. memory) to execute the processes.

2.7 Software Development

Tools to facilitate the development of NetKernel programs and the associated development cycle will be considered. Associated with the tools are a number of description formats used for input and output. The development cycle involves two phases: construction and mapping. Construction encompasses formulating a software network and supplying the code for its processes. Mapping, as described earlier, allocates a software to a physical network for execution. A fundamental benefit of the approach described in this chapter is that it provides a streamlined approach for mapping concurrent programs to parallel hardware. The actual program, expressed as a software network and associated code, need not be modified in order to change the mapping.

2.7.1 Description Files

The development tools use and produce various descriptions files. These files relate to various aspects of the paradigm and architecture:

F1 Process Type Description (PTD),
F2 Software Network Description (SND),
F3 Physical Network Description (PND),
F4 Mapping Description (MD).

2.7.2 Host Interface Program (HIP)

HIP is the most important of the development tools. It allows for runtime control of processes on a physical network. Some typical operations are loading, starting, stopping, and connecting outlets to input queues. These commands may be executed interactively or from scripts. Another tool, the script generator, creates script files for loading software networks.

2.7.3 Construction

The construction of a software network is broken into two parts. The first is a high-level description of the structure of the software network. This is referred to as the Software Network Description (SND). It declares the processes of the software network and their interconnection.

The second part of constructing is coding. A Process Type Description (PTD) and sequential program must be supplied for each type of user process in the software network. These may be reused in different NetKernel programs. The avoidance of connectivity information in the code of programs allows such reuse.

2.7.4 Mapping

Once a software network has been constructed, it may be executed on an available physical network. A Physical Network Description (PND) should be developed which describes the configuration of the hardware in the physical network. A mapping description (MD) is then developed which describes how the software network is mapped to the physical network.

Different MDs can be used to accomplish various mappings of a software network. Presently mappings are selected manually. Software tools to assist in selecting an appropriate mapping may be developed in the future. A software network may also be mapped to different physical networks. A PND is required for each different physical configuration. Software tools to identify the good configurations of physical hardware for a particular software network would also be useful.

There is no need to iterate between construction and mapping in order to explore different mappings. Some iteration, however, can be considered normal during development of software as it is tested and refined.

2.8 Simplicity and Adequacy of the Paradigm and Architecture

In the interests of simplicity, we have tried to limit the paradigm and architecture to a small number of basic concepts. The basic concepts of the paradigm are listed below.

- process
- software network
- input queue
- outlet
- connection
- message

The paradigm also defines a small number of primitives for message passing: SEND, REPLY, EXAMINE, WAIT, and TAKE. The architecture adds the following concepts.

- physical network
- station
- I/O device
- link
- mapping

The adequacy of the message passing primitives is considered in comparison to the paradigms of other environments in Chapter 5. The adequacy of the paradigm and architecture for achieving the operational requirements of Section 2.2 will be informally demonstrated by test programs in the appendices. Finally, the ability to support a higher level net-oriented programming paradigm based on a Petri Net design notation will be considered in the context of future research in Section 6.2.1.

Chapter 3 NetKernel: Design and Implementation

3.1 Overview

The NetKernel paradigm and architecture presented in the previous chapter have been implemented for Motorola 68000 based computer units. This chapter presents the design of the kernel software, as well as some information on implementation. Prior to the M68000 implementation, a prototype of NetKernel was developed which runs on NeXT workstations. The design of the prototype is described briefly at the end of the chapter, illustrating how NetKernel may run on stations ranging in complexity from simple microprocessor systems to workstations.

The design decisions presented in this chapter reflect particular approaches to realization of the NetKernel paradigm. Other approaches are certainly possible. For instance, static source routing was selected to route messages. Other techniques could have been used.

NetKernel provides a concurrent processes programming model for parallel hardware. In particular, Netkernel provides process management and interprocess communication across a physical network. Each station runs its own copy of Netkernel. Netkernel has been designed and implemented using the microkernel approach. The microkernel is a core which provides some basic facilities and services for the execution of processes. A collection of kernel processes present at start-up provide further services. The actual kernel processes present may vary from station to station due to differing processor types and peripheral devices. Services may also be varied at runtime by loading and unloading of processes. Stations of a particular type will usually run identical versions of NetKernel, although this is not required.

The microkernel provides the following basic facilities and services:

G1 message passing,
G2 process scheduling,
G3 memory management,
G4 exception handling,
G5 system clock activities.

The message passing system realizes the NetKernel concept of process outlets which are dynamically connected to input queues at runtime. A straightforward priority-based scheme was adopted for process scheduling. Scheduling is driven by the occurrence of exceptions and the message passing activities of processes. Memory management and exception handling are relatively minor aspects. Memory management is limited to allocation/deallocation of physical memory blocks. A special kind of process is used to service exceptions.

A benefit of the microkernel design is that kernel services can be added or deleted by adding or removing kernel processes. Kernel processes currently address the following areas:

H1 message routing,
H2 device servicing,
H3 process management,
H4 command line interface to the message system.

Message routing is performed using source routing through a store-and-forward network. A serial channel device driver supports both protocol-based passing of messages over network links, and a line oriented I/O interface to devices such as terminals and workstations. Process management includes creation, suspension, and profiling of processes. The command line interface allows dialogue of text messages with processes in the network. The decomposition into processes and a description of each process is presented later in the chapter.

In the following section, the design philosophy of this project is presented. Next, the hardware is described. The following section presents the high-level design, including a summary of each kernel process and the major functions of the microkernel. The following two sections elaborate on the kernel processes and microkernel, respectively (these two sections should be skipped if the reader is not interested in such details). A short, informal treatment of correctness is then presented. Finally, some implementation details are mentioned.

3.2 Design Philosophy

The design and implementation of a distributed system's kernel is a large project. To facilitate completion of this project within a reasonable period of time, a suitable design philosophy was adopted. The philosophy is to use simple mechanisms when such mechanisms do not compromise the operational goals laid out in the preceding chapter.

This philosophy is intended to avoid expenditure of excessive effort on non-central aspects of NetKernel. Areas requiring enhancement may be identified upon completion of the initial version and addressed in future work.

3.3 The MPAXS Hardware

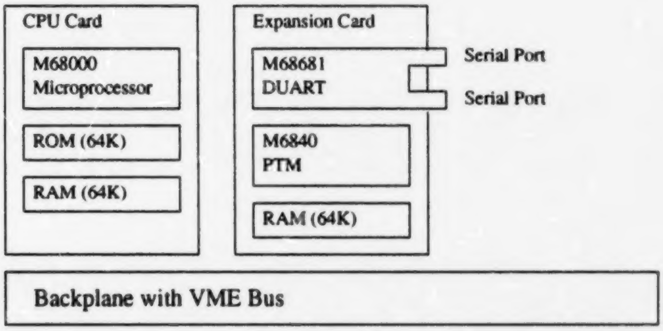
The MPAXS hardware is an assortment of computer systems available in the department. The computers are interconnected using communication links to realize physical networks. Regular and irregular topologies are both possible. A specific configuration is described by a Physical Network Description (PND). Departmental hardware presently being used for MPAXS will now be presented. The level of integration currently achieved with the NetKernel paradigm for different types of hardware will also be indicated.

3.3.1 Processors

Computer	Classification
M68000 unit	NetKernel Target (Station)
Sun-3 workstation	Non-NetKernel Host (I/O Device)
NeXT workstation	Prototype NetKernel Host (Station)

Table 1 MPAXS processors and their classification

Three types of computers are presently being used. These are listed in Table 1 along with their classification under the scheme of Section 2.5.1, "Stations".



DUART = Dual Universal Asynchronous Receiver/Transmitter
PTM = Programmable Timer Module

Figure 11 Hardware of the M68000 target station

Relatively simple Motorola 68000 microprocessor systems are being used for target stations. These units are also used in a stand-alone configuration for a microprocessor lab in the department. Each unit has a VME bus backplane holding at least two cards. A block diagram of the system is shown in Figure 11. A total of 128K RAM and 64K ROM is available in the basic unit. The DUART provides two serial ports, which are used to interconnect units. A speed of 9600 baud is being used for the serial ports. An additional expansion card can be added, providing an additional two serial ports and 64K RAM. Software available as a starting point for this work includes an assembler level debugger and a C cross compiler for programming.

Sun-3 workstations are used to run the development tools. This includes the Host Interface Program (HIP). HIP facilitates downloading object code to targets, and provides runtime control of processes. NetKernel has not been implemented to run on the Suns. Hence, the Sun workstations are included in the physical network as I/O devices. Each Sun has two serial ports for interfacing to the physical network.

A prototype of the kernel was developed which runs on NeXT workstations. NeXTs linked together by Ethernet are supported. As well, NeXTs have two serial ports which may be interfaced to M68000 target stations. This allows for the same physical network to contain two different station types (i.e. NeXT and M68000), as well as Suns used as I/O devices.

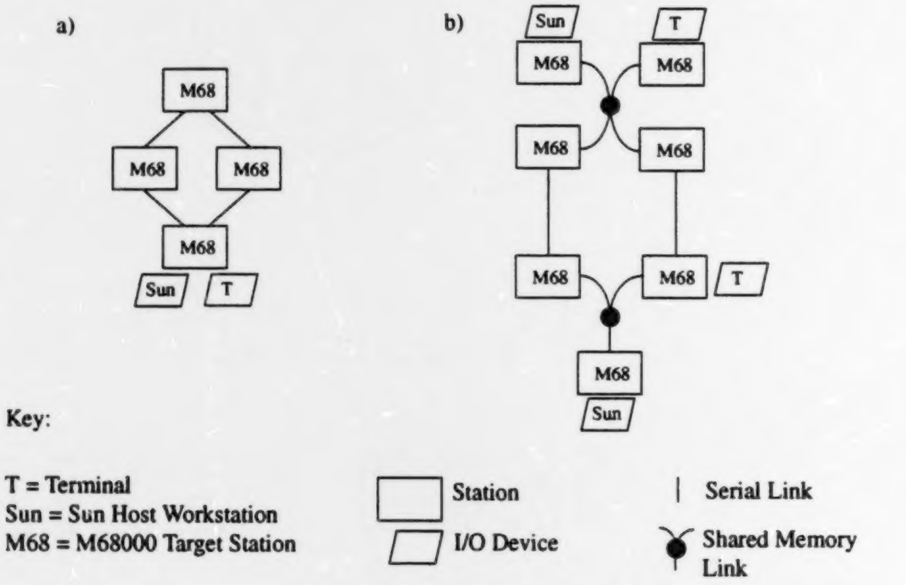


Figure 12 Examples of physical networks constructed from MPAXS hardware
a) Using serial links only (same configuration as test case in Section C.2).
b) Future physical network using both serial links and shared mem-

3.3.2 Physical Network

The hardware described above allows physical networks to be constructed by interconnecting M68000 stations with serial lines. Up to four serial ports are available per M68000 station. Sun-3 workstations may be interfaced to M68000 stations as I/O devices. Note that a single Sun-3 is sufficient for control of an entire physical network. An example physical network using available hardware is shown in Figure 12a.

Work on implementing a shared memory subsystem for the M68000 stations is underway at the time of writing. The shared memory will be used to link two to four M68000 stations. The NetKernel design considers both point-to-point links and shared access links. Hence, shared memory can be readily incorporated when it becomes available. Figure 12b shows an example of a physical network containing two clusters of stations, each cluster having a shared memory.

3.4 High-Level Design

This section provides an overall picture of the interface and design of the kernel. The kernel at a station runs concurrently with user processes, as shown in Figure 13a. An interior view of the kernel itself is shown in Figure 13b. It consists of the microkernel and a number of kernel processes. Kernel processes are similar to user processes. The difference is that kernel processes are automatically loaded and started at power up, whereas user processes are loaded and executed dynamically at runtime.

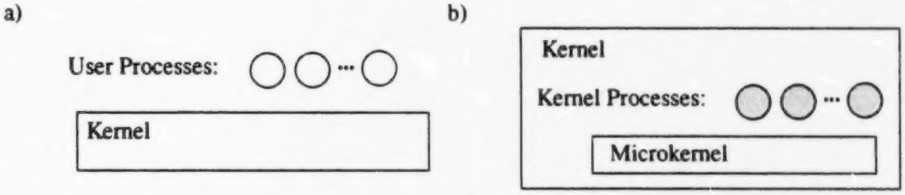


Figure 13 High-level views of the kernel
a) External view. b) Interior view.

A summary of the kernel's operation follows. Next, kernel processes and associated services are summarized. The major functions of the microkernel are then considered. Finally, NetKernel's interface is described.

3.4.1 Summary of Operation

The CPU of a station spends most of its time executing processes. Occasionally, an exception occurs or a process performs a system call. This results in the microkernel being entered. The microkernel performs a few short operations in response and then exits, possibly switching to a different process. The highest priority ready process always gets to execute, while other ready processes wait in queues for their turns.

Processes executing at a station are of two types: kernel and user. Some of the kernel processes provide services directly accessible to user processes, whereas other kernel processes transparently perform support operations. Services include setting up, starting, and control of user processes. The major function which kernel processes provide transparently is the distributed message system.

3.4.2 Message System

Design issues for the message system are routing, flow control, congestion control, and error control. These are major areas in the study and operation of computer networks. Consistent with the design philosophy, simple approaches have been taken.

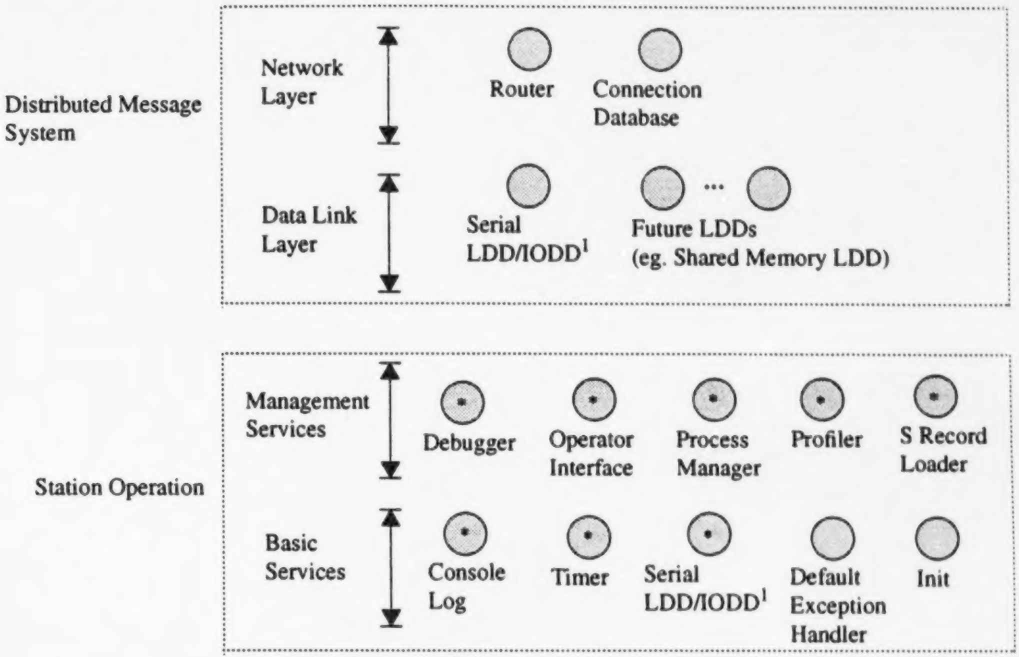
The software of the message system consists of two layers. These correspond to the data link and network layers of the Open Systems Interconnection (OSI) reference model [Day83]. Note that only some of the OSI functions associated with the two layers are actually present. The data link layer handles the transmission and reception of packets over individual physical links. A reliable, ordered packet service is provided across links. Specific error control may vary with link type. For the serial lines, backward error recovery is used in conjunction with checksums.

The network layer handles only the routing of packets through the physical network. An unacknowledged datagram service is provided. The size of a datagram is limited to the packet size. The technique of source routing is used. When an outlet is connected to an input queue, a route must be specified. All messages sent through the outlet will follow that route. The use of a static route and first-come-first serve queueing of packets along the route has the result that messages arrive at the destination in the order that they are sent. No mechanisms for dealing with congestion are provided. Nor does the network layer perform any error control. It is strictly best-try. Two approaches are possible to avoid losing messages due to congestion. The first approach is to map the software network in such a way that the physical network can keep up with the demand placed on it. When this is done, the reliability of the network layer is directly determined by that of the underlying data link layer. The second approach is to build mechanisms for dealing with congestion on top of NetKernel.

No transport layer is provided. So there is no end-to-end error control or flow control. The lack of end-to-end error control is considered acceptable here since the data link layer is reliable. This avoids duplicating error control in different layers. To avoid losing messages due to a fast sender overrunning a slow receiver, the approaches of suitable mapping or user level mechanisms are again appropriate.

One additional consideration in the design of the message system will be mentioned. In order to facilitate mapping to a wide variety of hardware, it should be easy to add support for new types of links to the message system. This has been accomplished by associating a device driver process with each link type. Incorporating a new type of link simply requires adding a new device driver to the kernel.

3.4.3 Kernel Processes



* indicates a process which provides user accessible services.
LDD = Link Device Driver, IODD=I/O Device Driver
[1] Serial LDD/IODD has two roles, one as a data link process (LDD) and one as a basic service (IODD).

Figure 14 Kernel processes of the M68000 version of NetKernel

Each M68000 station has an identical suite of kernel processes. These processes are activated from ROM upon start-up. Specific processes have been selected to support the paradigm and architecture presented in Chapter 2. These are illustrated in Figure 14. For simplicity, only a bubble is drawn for each process. Input queues, outlets, and connections are not shown. The criteria of [Gomaa84] were used to make the process decomposition. These criteria include dependency on I/O, time-critical functions, functional cohesion, temporal cohesion, and periodic execution.

The kernel processes have been divided into major groups according to function. One group is concerned with operation of individual stations. These processes are involved in local affairs of the station on which they run. The other group supports the distributed message system. The functions of processes from both groups are summarized in Table 2.

Kernel processes which provide services directly accessible to user processes have been marked with an '*' in both Figure 14 and Table 2. A user process accesses a service by

Process	Description
Distributed Message System	
Connection Database	Records connection information for process outlets, including routes for nonlocal connections.
Router	All incoming and outgoing packets pass through Router. If an incoming packet has arrived at its destination station, then Router places the arrived message into its destination input queue. Otherwise, the packet is sent to the Link Device Driver (LDD) for its next hop.
Serial LDD/IODD (LDD role)	Manages the serial ports of a station. Each port is in one of two modes: link mode or I/O mode. Ports in link mode (LDD role) are used to exchange packets with other serially connected stations.
(Other LDDs)	May be added in the future to support new link types.
Station Operation	
Debugger*	Provides a low-level debugging service.
Default Exception Handler	Responds to unexpected and debugging exceptions.
Init	Involved in system start-up.
Log*	Allows messages to be sent to a local and central log.
Operator Interface*	Provides a command line interface to the message system. This allows an operator or development tool to dialogue (i.e. send messages and receive replies) with any process in the physical network.
Process Manager*	Central process in managing station operation. Commands are sent to Process Manager to set up, start, interconnect, and otherwise control the execution of processes at a station.
Profiler*	Reports statistical profiling of execution of processes.
S Record Loader*	Loads object code from a host into the memory of an M68000 target station.
Timer*	Provides timer services. One-shot, periodic, and shared timers are available.
Serial LDD/IODD (I/O role) *	Manages the serial ports of a station. Each port is in one of two modes: link mode or I/O mode. Ports in I/O mode (IODD role) are used to input and/or output data with serial I/O devices.
* indicates a process which provides user accessible services	

Table 2 Summary of kernel processes

exchanging messages with the corresponding kernel process. The two groups of services will be considered briefly. More details on services and the kernel processes which implement them are provided in Section 3.6.

Most of the station operation processes provide services. These have been divided into two groups: management services and basic services. Management services include setting up, controlling, and monitoring the execution of processes at a station. These can be accessed interactively by an operator or by higher level software such as development tools. Basic services are useful to the processes running at a station, but are of little interest for management of the station.The processes Init and Default Exception Handler have been shown in this group even though they provide basic support for the kernel rather than user accessible services.

The distributed message system processes are largely transparent in the operation of NetKernel. They do not provide services which are directly accessed by user processes or the operator. Packets are received, routed, and transmitted transparently. When a process sends a message to a nonlocal destination, the microkernel transparently passes the message to Router.

The Serial LDD/IODD process is unusual in that it has two roles. In one role, it is a transparent component of the distributed message system. In the other role, it acts as a station operation process which provides a service for serial I/O. The use of a single process for both roles was preferable here for low-level implementation reasons. Each DUART has two serial ports which may be used independently. In the case where one port is being used as a link and the other for I/O, it would have been awkward to coordinate the access of two processes to the hardware registers of a single DUART. Otherwise, two processes would have been used. Command messages are sent to Serial LDD/IODD to select the mode for each serial port.

3.4.4 Microkernel

The five major responsibilities of the microkernel were listed earlier as items G1 through G5. The approach taken for each of these is summarized below. All system calls are serviced by the microkernel. System calls support message passing, memory management, some other miscellaneous functions.

Process scheduling is preemptive based on priority, but it is not round robin. The microkernel uses a Process Control Block (PCB) to record information about each process. This includes its state, priority, a Condition expression when it is blocked waiting (Section 2.4.2), and other information. A queue of ready processes is maintained for each priority. The highest priority ready process executes until it gives up the CPU by performing a blocking WAIT, or until it is preempted by a higher priority process becoming ready. A higher priority process can become ready due to the arrival of a message or an interrupt for it. Round robin scheduling of some or all priorities could be added, but this has not been done yet.

The message passing system calls of the microkernel, together with the Router process, constitute the network layer of NetKernel. When a message is sent, the microkernel determines whether it is being sent to a local or nonlocal process. If the destination is local, then the message is immediately deposited into its destination input queue. If the destination is nonlocal, then the message is put into a network packet and deposited into an input queue of the Router process. The network packet has a header specifying the destination process and queue, as well as the route to be taken. Once a message has arrived at its final destination, the kernel deposits the message and a reply key into the receiver's input queue. The reply key may be used by the receiver to send reply messages if the original sender indicated that replies are allowed.

The M68000 target stations do not have any memory management hardware. All processes execute in a single address space. As a result, there is no protection between processes. A malicious or erroneous process can disrupt the kernel or other processes by writing to memory for which it has no rights. A scheme for memory allocation/deallocation has been devised so that well behaved processes can co-exist in the same address space. There are two levels to this scheme. At the first level, physical memory is divided into regions which are owned by processes for code, data, and stack. This level is actually the responsibility of the Process Manager. At the second level, the microkernel dynamically allocates blocks of memory to processes making Allocate() and Free() system calls. Blocks are limited to a fairly small size to simplify memory management. A message arrives into the address space of a process as a new block. The block should be deallocated by the process when it is no longer needed.

All exceptions, including interrupts, cause the microkernel to be entered. The microkernel responds by either servicing the exception itself or informing a process which services the exception. An example of an exception serviced by the microkernel itself is the clock interrupt. Exceptions not serviced by the microkernel are serviced by a special type of process, referred to as a *handler process* or simply handler. Handlers are subject to certain restrictions compared to ordinary processes in order to improve the efficiency of exception servicing.

The various station activities driven by the clock interrupt are part of the microkernel. These could have been realized using one or more kernel processes. Instead, to reduce overhead, routines directly in the microkernel were used. The following activities are performed:

I1 statistical profiling of execution of processes,
I2 suspension of high priority runaway processes,
I3 updating of station time and timers of the Timer process.

3.4.5 Interface

NetKernel presents two distinct interfaces. The first of these is the system call interface of the microkernel. The second is the message interface presented by kernel processes.

The message passing primitives described in Section 2.4.2 have been realized as system calls. Some refinements of the Chapter 2 presentation have been made. A process also obtains a *reply key* whenever it removes a message from an input queue. The reply key may used by the receiving process to send reply messages if the original sender indicated that replies are to be accepted. Function prototypes for selected system calls are included in Section B.1.

Each of the services mentioned earlier has an individually defined message interface. Each interface specifies the format of messages exchanged with the associated kernel process. An ASCII format has been used for all messages. This has the benefit that an operator may access any of the services interactively through the Operator Interface process using a terminal. Process management commands, for instance, can thereby be issued to any station from a single terminal of the physical network. The development tools may also issue commands in the same manner, to provide a higher level interface from workstations. The specific syntax of the messages will not be discussed here, but as an example the interface for Process Manager is included in Section B.2.

3.5 Microkernel Design

The microkernel is intended to be small and fast. Most of the kernel's complexity is contained in kernel processes, meeting the goal of small size. A top-down procedural approach to the design of the microkernel has been manageable due to its small size. The two main aspects of the design are its procedure hierarchy and data structures.

The kernel data structures contain information about a station's state. This information needs to be maintained in static variables as opposed to local variables of procedures, because it must be persistent between entries to the microkernel. To improve the design by reducing data coupling between modules, the number of modules which access each data structure was kept to a minimum.

The procedures of the microkernel are grouped into a collection of modules. These are introduced in the following subsection. A discussion of access to global data structures follows. The responsibilities and design of each module are then considered individually. Important global data structures are described where appropriate.

3.5.1 Module Summary

A module has been associated with each of the microkernel responsibilities listed in items G1 through G5. The modules contain procedures, microkernel entry points, and microkernel exit points. The call structure of the modules is shown in Figure 15. Each module calls procedures in other modules which are connected directly beneath it.

The figure also shows entry points into the microkernel. There are three types of entry into the microkernel: system calls, processor exceptions, and the clock interrupt. There is only one exit point from the microkernel, which is from the Process Scheduling Module. Hence all of the other modules exit through this point. Upon exit, control is either returned to the previously executing process, or given to a different process.

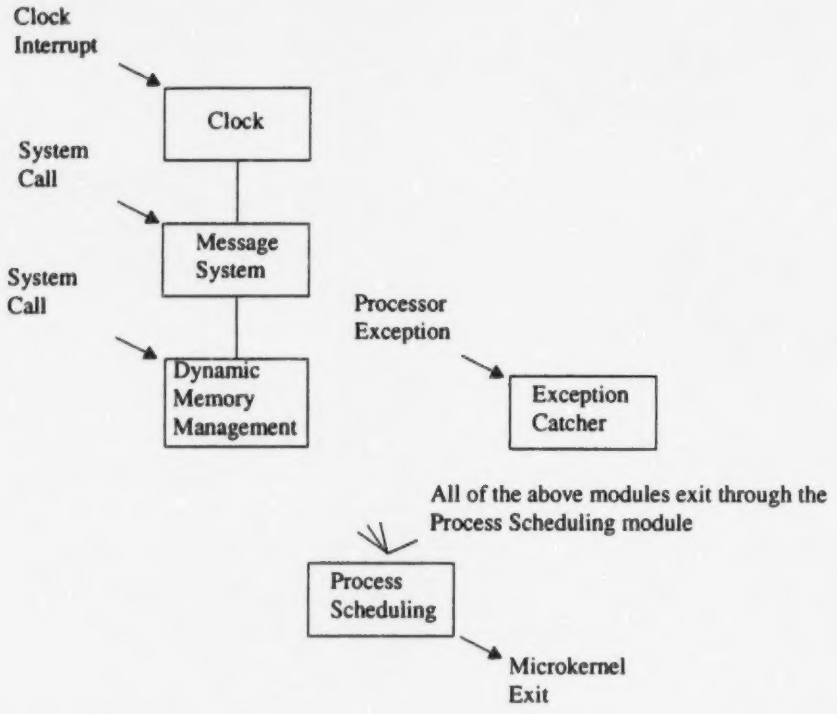


Figure 15 Structure chart of microkernel modules showing entry/exit points

3.5.2 Kernel Data Structures

The kernel data structures provide a mechanism for the microkernel and the kernel processes to exchange information and influence each others' behavior. From a design perspective, this could have been accomplished more elegantly by using only the following mechanisms:

J1 system call parameters for information from kernel processes to the microkernel,

J2 system call return values and messages for information from the microkernel to kernel processes.

Indeed, system calls are the only mechanism permitted for exchange between user processes and the microkernel. It would be highly undesirable for user processes to directly access kernel data structures since this would introduce data coupling and makes the kernel more vulnerable to erroneous user processes. These concerns do not apply to the same extent for kernel processes. Data coupling within the kernel is less of a problem than data coupling between the kernel and user processes. We have more confidence in the correctness of kernel processes than user processes, so vulnerability of the kernel is also less of a concern. Hence data coupling has been used to achieve a more efficient implementation. It should not be used excessively and needs to be well documented.

The following subsection summarizes the most important kernel data structures. Next, the need for mutual exclusion and how the kernel achieves it are described.

Name	**Description**
Dynamic Memory Map (DMM)	Used to manage allocation of dynamic memory blocks.
Connection Block (CB)	Holds the destination information, including route, for one connection. Each connected outlet of a process has its own CB.
Exception Control Block (ECB)	An ECB is associated with each type of M68000 exception. The ECB indicates a handler process and contains exception statistics.
Process Control Block (PCB)	A PCB is associated with each process. It contains information on state, resource usage, parameters, and statistics of the process.
Diagnostic Block (DB)	This block records diagnostic information which may be useful for post-mortem debugging following a kernel crash
Station Control Block (SCB)	Holds various aspects of a station's state, including resource usage and the process queues for scheduling.
Timer Control Block (TCB)	A TCB is allocated for each timer in use at a station. Holds timer parameters and state.

Table 3 Summary of kernel data structures

3.5.2.1 Summary of Structures

The kernel data structures hold information on the state of the kernel and the processes being managed by the kernel. Table 3 summarizes the kernel data structures. These are accessed by both the microkernel and kernel processes. To reduce data coupling throughout the design, the number of microkernel modules and kernel processes accessing each data structure has been kept to a minimum.

3.5.2.2 Mutual Exclusion

Multiprocessing systems require mutual exclusion of access to shared data. This is discussed in most operating systems texts such as [Silbersch89] and [Tanenbaum92]. To illustrate the need, consider two concurrent processes A and B which must both increment a shared variable *counter*. The execution of these processes may interleave such that the following sequence of operations occurs.

- A reads counter
- B reads counter
- A increments its version of counter
- B increments its version of counter
- A writes its new value
- B writes its new value

Upon completion, counter will have been increased by one instead of two. The solution to this and other similar errors is to provide a general mechanism which allows a process to obtain exclusive access to some data while operating on it. Given such a mechanism, the execution of processes A and B could no longer interleave in such a way as to produce an incorrect result.

Mutual exclusion is a concern for NetKernel since the microkernel and kernel processes concurrently access the kernel data structures. The approach taken in NetKernel is to protect these accesses in critical sections. Upon entry to a critical section, the CPU priority is raised to its maximum level, blocking out all interrupts which could possibly result in a violation of mutual exclusion. Upon exiting the critical section, the CPU priority is restored to its previous value. [Tanenbaum92, p. 35] states that disabling interrupts is a useful technique for kernels, but is generally dangerous for user processes.

The use of priority to realize critical sections is quite efficient in that it imposes almost no synchronization overhead. It is also attractive from the perspective of avoiding deadlocks. Since processes are never blocked to wait for a data structure, this mechanism cannot cause the kernel to deadlock. The alternative mechanism of semaphores has greater overhead and requires care to avoid deadlocks. A disadvantage of blocking all interrupts in comparison with the use of semaphores is that absolutely no other activities may take place concurrently, including higher priority interrupts which do not conflict with the critical section. To minimize the blocking of other activities, critical sections should be kept as short as possible.

A common source of difficulty when dealing with concurrent processes is to ensure that all the appropriate data accesses are protected. Omitting protection of an access can lead to timing dependent errors which are hard to detect and reproduce. To avoid such problems, a fairly sweeping approach has been adopted. All entries to the microkernel are blanketed within a critical section. The critical section ends upon exit from the microkernel. A similarly comprehensive approach has been used for kernel processes. This has the benefit of providing full coverage of accesses with little effort. However, it is overkill, producing critical sections much longer than necessary. This conflicts with the need to keep critical sections short in order to avoid blocking other activities. This compromise has been accepted for the moment since it simplifies the kernel and reduces the likelihood of programming errors.

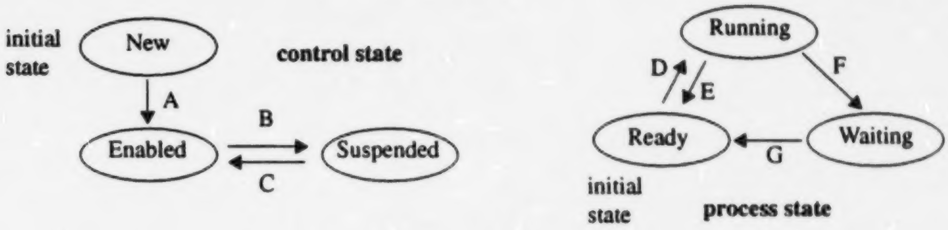
3.5.3 Process Scheduling

The scheduling policy has been described in Section 3.4.4. The mechanism used to implement the policy will now be presented. First, the two state variables of a process which affect scheduling are presented.

3.5.3.1 Process State and Control State

For every process, two state variables related to scheduling are maintained in the Process Control Block of the process. These are referred to as the *control state* and *process state* of the process. Figure 16 shows the corresponding state diagrams. The control state is influenced by process control operations, such as start and suspend commands to Process Manager. The process state is influenced by the process's own behavior (eg. waiting for a message) and the behavior of other processes (eg. arrival of a message from another process, preemption by a higher priority process). When a process is first allocated, its control state and process state are initially New and Ready, respectively.

Note the distinction between Waiting and Suspended. A Waiting process is blocked from executing while it voluntarily waits for some input. A Suspended process is prevented from executing as a result of some external or unexpected event, such as a debugging breakpoint or zero divide exception. Since the two state variables are largely independent, it is possible for a process to be both Waiting and Suspended at the same time. The only coupling between process state and control state is expressed in state change D of Figure 16.



A. Start command for newly allocated process is issued to Process Manager
B. Process is suspended due to an exception or by a command to the Process Manager or Debugger.
C. Process is continued by a command to Process Manager or Debugger.
D. Process is selected by the microkernel to execute. Occurs only when control state=Enabled.
E. Process is suspended or preempted.
F. Process becomes blocked to wait for input, i.e. message(s) or interrupt.
G. Input for which process was waiting arrives.

Figure 16 State diagrams for process state and control state

3.5.3.2 Scheduling Mechanism

Scheduling is achieved by keeping processes which are both Enabled and Ready in *process queues*. There is one such queue for each priority level. The microkernel selects the process to run by taking the front process of the highest priority process queue which is nonempty. If a process is preempted by a higher priority process, then the preempted process is returned to the front of its queue.

The microkernel checks to see if the Running process should be preempted every time that there is an exit from the microkernel. This is implemented in the Process Scheduling module, which contains the sole exit point of the microkernel. It will select a new process to become the Running process[1] if any of the following events have occurred:

K1 A process queue of higher priority than the Running process has become non-empty,

K2 The Running process has become blocked waiting for input,

K3 The Running process has been suspended,

K4 An exception has occurred.

These events always result in an entry to the microkernel, which guarantees the opportunity to select a new process for execution upon exit. To illustrate, consider item K1. One situation in which this event occurs is when the Running process does a Send() to a local process of priority higher than itself. If the receiving process is unblocked due to the arriving message, then it will be placed at the end of its process queue by the Send() system call. In this case, the microkernel will detect K1 upon exit from Send().

3.5.4 Dynamic Memory Management

The lack of memory management hardware and the basic approach to memory management were described in Section 3.4.4, "Microkernel". The Memory Management module of the microkernel provides system calls for allocation and freeing of small blocks of dynamically allocated memory. Consistent with the design philosophy, simplicity has been favored. The penalty here has been performance, utilization, and vulnerability to fragmentation, although the scheme is not excessively poor in any of these areas. The kernel also supports reservation of larger regions of memory for process code and data, which is handled by the Process Manager (Section 3.6.3.1).

Dynamic blocks are allocated from a single contiguous region of physical memory. This region is finely divided into *lines*. Currently, the line size is 32 bytes. The present maximum block size is 256 bytes which requires eight contiguous lines. This maximum also limits the length of messages, since each message must fit into a block. A kernel data structure, Dynamic Memory Map, holds a table indicating the usage of lines. Each entry indicates either 1) the process to which the line is allocated, or 2) that the block is currently unused. The scheme to select a contiguous group of lines to satisfy a request has been designed to reduce search time and fragmentation, but could still be improved. The small maximum block size also helps avoid fragmentation problems since it eliminates the need to find large contiguous sections of free memory.

The Memory Management module also provides procedures for use only by the kernel. The Transfer() procedure, for instance, is used to transfer ownership of a block between two processes at a station. This avoids copying when a message is passed between processes at the same station if the sender has indicated that it no longer needs the message block (indicated by the *freeFlag* parameter of the Send() system call). Another internal

---

[1] The Init process executes an empty loop at the lowest possible priority, so there is always at least one process available to execute.

procedure, FreeAll(), is by called Process Manager to free all the memory blocks owned by a process which has been killed.

The microkernel enforces a limit on the number of lines which a process may own. Limits may be specified individually for processes. A process which tries to exceed its limit is suspended. The dynamic memory usage of individual processes can be monitored using Profiler (Section 3.6.3.6).

3.5.5 Message System

As introduced earlier, the microkernel contributes to the network layer of the message system. The other component of the network layer is the Router process. The data link layer is realized by device driver processes. Messages passed between processes at the same station are handled entirely by the microkernel which avoids the overhead of going through intermediate processes. Passing of messages between processes at different stations involves the microkernel, Router, and data link processes at all stations along the route.

Here we focus on the contribution of the microkernel (kernel processes will be discussed later). The Message System module realizes the system calls related to messaging. Design aspects of four of the main calls are presented below. Of these calls, only Wait() is blocking. Since no flow or congestion control is provided, both Send() and Reply() are nonblocking. Function prototypes for selected system calls are provided in Section B.1, including the four calls below (for which the implementation names are MP0_send(), MP0_waitExpression(), MP0_take(), and MP0_reply(), respectively). Call parameters are not shown below, but are included in the prototypes of Section B.1.

3.5.5.1 Send()

The microkernel determines whether the destination process is local or nonlocal. This is determined by examining the Connection Block (CB) associated with the outlet through which the message is being sent. If the destination process is nonlocal then the route to the destination station is read from the CB. In both cases, a header is constructed which includes the following information:

L1 the route to the destination process, which is null if the destination is local,
L2 the destination process id and queue id,
L3 the sender's process id,
L4 the length of the message,
L5 the id of the sender's input queue into which reply messages are to be sent (a special value indicates that the sender does not wish to accept replies).

If the destination process was determined to be local, then the microkernel immediately deposits the message and header into the destination input queue. If the destination process is in the Waiting state, then the microkernel checks to see if it should be unblocked with the arrival of the message. If so, the destination process is added to the end of the appropriate process queue. Figure 17 shows the data path for a local message. It is quite

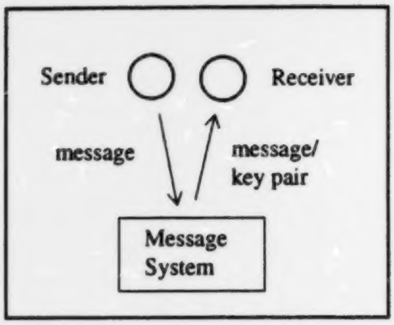


Figure 17 Data path and units exchanged for a local message

simple, involving no intermediate processes. Hence, local message passing involves relatively little overhead.

If the destination process was nonlocal, then the microkernel passes the message and header to the Router process. These will propagate through the physical network. Upon reaching the destination station, both are placed into the destination input queue. The data path for nonlocal messages will be illustrated later when kernel processes are discussed (Figure 21).

The reply key which the microkernel passes to the receiving process is actually just a pointer to the header for the message. The receiving process should never examine the header, but may use the key to send reply messages.

3.5.5.2 Wait()

Wait() is the only system call which may block the calling process. The input queues of the caller are tested to see if the Condition parameter is true. If true, then the call returns immediately without blocking.

If Condition is false, then Condition is recorded in the PCB of the caller. The caller is blocked by changing its state to Waiting. As for all other system calls, exit is through the Process Switching module. Since the caller has been blocked, a new process will be selected to run.

3.5.5.3 Take()

Take() removes a message and header pair from an input queue of the calling process. The caller is not actually entitled to examine the header, but rather considers it is as a reply key. It may be used by the caller to send replies to the message which was just removed.

3.5.5.4 Reply()

Reply is identical to Send(), except that it examines the reply key instead of the Connection Database. The reply key points to the header of the message to which a reply is being made. From this header, it can be determined whether the destination process is local or nonlocal. If the destination station is nonlocal, then the header also contains the route to the destination. Given this information, the call proceeds identically to Send().

The route to the destination was constructed in the header as it propagated through the physical network. This will be described further in Section 3.6.4.3.

3.5.6 Exception Catcher

The M68000 produces an exception in response to exceptional internal or external conditions.

> The exception may be internally generated by an instruction or by an unusual condition arising during the execution of an instruction. Externally, exception processing can be forced by an interrupt, by a bus error, or by a reset. [Motorola89, p. 6-1]

The kernel provides a response for all possible exceptions. The Exception Catcher module is invoked upon the occurrence of any of the M68000 exceptions apart from a few special ones. Exception Catcher responds by notifying a *handler process* of the exception. Handler processes are distinguished from ordinary processes in that the microkernel informs them of exceptions. They are implemented differently and subject to certain restrictions compared to ordinary processes. The default handler for most exceptions is Default Exception Handler. Exceptions can be individually associated with other handlers through setup commands to Process Manager. For more information on the default response to various exceptions, see Section 3.6.3.2, "Default Exception Handler".

The special exceptions which do not go through Exception Catcher are the system call trap, event log trap, clock interrupt, and reset. Each of these has its own individual entry point into the microkernel.

The microkernel should incur as little overhead as possible in exception servicing. This is particularly important with respect to interrupts, as discussed in the next subsection. The motivation for handling interrupts through the microkernel is then presented. Next, the differences between handler processes and ordinary processes are presented. The final subsection summarizes the classification of processes under NetKernel on the M68000.

3.5.6.1 Device Drivers

Devices attached to the VME backplane of a station are serviced using interrupts. As with other exceptions, interrupts are serviced by handler processes. A handler which services interrupts is referred to as a device driver. Handlers are allowed to perform interprocess communication using the message system similarly to ordinary processes. This makes the services of device drivers readily accessible to other processes through message passing.

The microkernel adds overhead to the processing of interrupts. This results from code in the microkernel that is executed for every interrupt. This microkernel code is executed in addition to the device driver code which actually services the interrupt. It is important that exceptions can be serviced quickly since interrupts can occur at high rates. Too much overhead will unacceptably limit the maximum rate. Another factor is that the servicing of an interrupt is often simple, such as reading a character and storing it in a buffer. Too much overhead results in a poor ratio of (device driver service time) / (microkernel overhead time).

3.5.6.2 Interrupt Servicing Through the Microkernel

The fastest and simplest method to service interrupts is to simply avoid going through the microkernel. However, certain aspects of this design require going through the microkernel:

M1 use of message system by service code,
M2 statistical profiling of exception handlers,
M3 recording exception entries and exits into the event log (to be introduced in Section 3.5.8.1),
M4 counting the number of exceptions.

The primary advantage of going through the microkernel is the use of the message system (item M1). The handler can then communicate with other processes by message passing. So from the perspective of other processes, the handler appears to be simply another process. Items M2 through M4 are less fundamental but useful elements of the design.

If the simple approach of avoiding the microkernel were taken, then an interrupt's vector would point directly to the code which services it. Upon completion, the service code would return to the context which was executing at the time of the interrupt. The disadvantage of this approach for NetKernel is that the service code would be unable to send messages. To support message sending, the service code should exit through the microkernel so that the context can be switched if necessary. A switch will be necessary if the service code sent a message which unblocked a process of higher priority than the interrupted process. The MCX-16 uniprocessor kernel for embedded systems, for instance, provides a "common Interrupt Service Exit Function" [Barrett92, p. 3.6] to determine if a context switch is required as a result of an interrupt.

In this design, we go one step further in that all exceptions both *enter* and *exit* through the microkernel. The microkernel performs operations associated with items M1 through M4 before and after the service code for an exception is invoked. Statistical profiling is supported by updating a kernel variable containing the pid of the currently executing process before service code is invoked (this variable is read by the Clock module upon clock interrupts). Calls may be made to record the start and end of exception processing in the event log. The total number of occurrences for each exception is updated in its ECB.

The specific mechanism used to realize service code, the handler process, is discussed next.

3.5.6.3 Handler Processes

Handler processes have been introduced to efficiently support M68000 exception processing within the NetKernel paradigm. Although the previous two subsections have focused on interrupts, recall that handlers also respond to other exceptions such as bus error or divide by zero. Two specific overheads have been avoided. First, no context is saved or restored for a handler between exceptions. Second, there are no system calls required on a per exception basis.

Handlers are M68000 specific. Other station types in the physical network need not support them. Indeed, other station types may provide different forms of support for integrating their exception servicing into the NetKernel paradigm. Exception handling is generally platform specific, so this does not introduce any new constraints on the mapping of processes.

A handler is not implemented in the same fashion as other processes. An ordinary process is implemented by a sequential program which may intermittently block waiting for input. A handler is implemented by an initialization procedure and a service procedure. Neither of these procedures are allowed to perform any blocking calls. For purposes of runtime control and from the perspective of other processes, however, a handler is treated identically to other processes. The same commands to Process Manager are used to start, kill, etc. Other processes may send messages through their outlets to handlers and receive replies back. Handlers also have outlets through which they send messages. Hence, we will still refer to handlers as "processes".

Exceptions are associated with a handler using setup commands to Process Manager. When a handler is started, its initialization procedure is invoked. Following initialization, its service procedure will be called by the microkernel each time an exception associated with the handler occurs. No context (i.e. CPU registers or stack) is restored or saved for the handler between calls to its service procedure. This reduces overhead and improves the response time to service an exception. Handlers may preserve state information between invocations using static variables for flags, counters, buffers and so on. Handlers run in privileged mode and share the same stack as the microkernel. Therefore, an erroneous handler is more liable to bring down the kernel (by not properly cleaning up the stack, for example). This makes handlers harder to develop and debug. A good design philosophy is to minimize the functionality in handlers by using additional ordinary processes where acceptable. Ordinary processes have the advantages of being easier to debug and develop since they are not subject to the restrictions of handlers.

There are no system calls that must be made by the service procedure on a per interrupt basis. Requiring a system call such as Wait() for every exception would have introduced substantial overhead. When the service procedure is finished, it simply executes a return instruction. Note that the service procedure is allowed to perform system calls to send messages, allocate blocks, etc. But the service procedure of a device driver with high interrupt rate will typically make such calls on only a small fraction of its invocations.

Handlers are not allowed to perform calls which block. Blocking calls require context to be saved for a process while it is blocked, which is being specifically avoided for handlers. Forbidding blocking calls also avoids the problem of what should be done if a device driver is blocked waiting for something else when its interrupt occurs. The restriction on blocking calls required a new mechanism (other than Wait()) to inform handlers of arrived messages. Essentially, a handler can specify that its service procedure should be invoked upon the arrival of messages into selected input queues. This is in addition to invocation upon associated exceptions. Two system calls, Enable() and Disable() are provided for this purpose (prototypes in Section B.1). This mechanism is not as convenient or powerful as the blocking Wait(), but has been adopted to facilitate simple and efficient handlers.

3.5.6.4 Process Classification

	Kernel	User
Ordinary	Any processing other than exception servicing.	Any processing other than exception servicing.
Handler (M68000 only)	Exception servicing only.	Exception servicing only.

Figure 18 Process classification and functions

The types of processes are summarized in Figure 18. To review, kernel processes are started automatically at start-up as part of NetKernel's operation. User processes may be loaded and started dynamically at runtime. Most processes are ordinary processes. Handlers are a special type of process which are used on M68000 stations to service exceptions. Handlers have a special structure consisting of initialization and service routines, and are restricted from performing the Wait() call since it can block.

3.5.7 Clock

The Clock module is driven by clock interrupts to perform a number of periodic activities. These were presented as items I1 through I3. The clock interrupt repeats at a fixed interval. The activities of the clock module are performed at various multiples of this interval.

The intervals currently used are shown in Table 4. These may change with further experience and study. The various activities of Clock are discussed below.

3.5.7.1 Statistical Profiling

Statistical profiling is performed by periodically recording which process is executing. The purpose of this activity is to determine the approximate CPU utilization of the processes at a station. The clock module maintains two tables. The first indicates how many times each process has been recorded in the current *profiling interval*. The second table

Name	Period (ms)	Rate (1/s)
Clock Interrupt	5	200
Statistical Profiling	5	200
Runaway Process Suspension	5	200
Updating Station Time and Timers	50	20

Table 4 Periods of system clock activities

indicates how many times each process was recorded in the previous profiling interval. A profiling interval of 200 clock interrupts, which equals one second, is being used presently. A lifetime total for each process is also recorded in its PCB. Profiling information can be reported using the services offered by Profiler and Debugger.

3.5.7.2 Runaway Process Suspension

User processes may be placed at priority higher than kernel processes in order to ensure good response time. High priority user processes should not perform excessively lengthy operations since kernel processes are prevented from running during such operations. In the worst case, a high priority user process can erroneously enter an infinite loop. When this happens, the process is referred to as a runaway process. A runaway process prevents any reactive control by other processes, since they never obtain the CPU. Even the operator cannot send commands to the station because the link device drivers do not get any CPU time. Hence, a mechanism to suspend runaway processes has been included in the clock module. Whenever the priority of the CPU does not fall below a predefined level for a predefined length of time, the next user process interrupted by a clock interrupt will be suspended. This mechanism is effective because the priority of the clock interrupt is high enough to interrupt all processes [1].

3.5.7.3 Updating Station Time and Timers

The time since start-up is maintained by the Clock module. The Clock module also assists the Timer process with the provision of timer services. Clock decrements countdown variables in Timer Control Block data structures which Timer sets up. Whenever one of the countdown variables reaches zero, Clock sends a message to Timer. This approach avoids the overhead of having to perform many context switches to Timer in order to simply decrement timers.

As Table 4 indicates, time is updated less frequently than other clock activities. This was done to reduce the likelihood of a critical section blocking the clock interrupt for so long that the station time falls behind. Updating of timers is a fairly lengthy operation, so performing it less frequently also reduces microkernel overhead.

---

[1] The clock interrupt does not interrupt a process in a critical section. Hence a process which goes runaway within a critical section can hang up its station. Fortunately, critical sections are not ordinarily found in user processes.

3.5.8 Miscellaneous

This section describes minor aspects of the microkernel design which did not fit into any of the preceding subsections.

3.5.8.1 Event Logging

The microkernel supports recording event occurrences along with a timestamp into an *event log*. This can be used to provide a trace of events to assist in debugging. It can also be used to assess software performance through timing measurements. The kernel currently supports logging a number of standard event types including exception entries and exits, system call entries and exits, and context switches. The logging of each event type can be turned on and off individually. The user can also define additional event types by introducing trap instructions, referred to as *event log traps*, into user code.

Each log entry includes a timestamp which is read from a free-running counter of a peripheral chip (i.e. the M6840 Programmable Timer Module). The recording of each event into the log introduces an overhead of approximately 100 microseconds. This overhead changes the timing of processes, making the event log inadequate for studying errors which are sensitive to exact timing. Nonetheless, the event log can be useful for detecting errors not related or less sensitive to timing,

An example of an event log report produced by one of the development tools is included in Appendix E.

3.6 Kernel Processes

Kernel processes perform two types of activities: transparent activities and user accessible services. Examples of each type are packet switching and timer services, respectively. The individual kernel processes along with their functions and services were introduced in Section 3.4, "High-Level Design". In particular, Figure 14 showed the organization of the kernel processes into Network Layer, Data Link Layer, Management Services, and Station Services. This section describes the interactions between kernel processes and their individual designs.

First, general aspects on the design of the kernel processes are presented. Next, the kernel processes are presented in two groups, corresponding to the major division of Figure 14. One group performs individual station activities and the other realizes the distributed message system.

3.6.1 Interaction Between Kernel Processes

The standard mechanism for exchanging information between processes is message passing. This is the only mechanism which NetKernel directly supports for user processes. In addition to message passing, kernel processes exchange information through shared access to kernel data structures. This improves the performance of the kernel by avoiding the overhead of message passing.

The kernel data structures were introduced in Section 3.5.2, "Kernel Data Structures". Mutual exclusion is required between kernel processes and also between kernel processes and the microkernel. Critical sections are used to provide mutual exclusion. The sweeping approach of placing large amounts of kernel code in critical sections has been adopted. As discussed in Section 3.5.2.2, this approach is taken for reasons of simplicity and coverage, but does compromise the response time for high priority processes.

3.6.2 Process Decomposition

This subsection describes the decomposition of the kernel into individual processes. A useful result of process decomposition is that response time to various events can be influenced by the arrangement of process priorities. The relative priorities of the kernel processes will also be presented.

3.6.2.1 Considerations

Each of the processes in the station operation group (Figure 14) is largely independent of the others in terms of the service or support it provides. The activities of this group were decomposed into separate processes based on the following criteria of [Gomaa84]:

- response time criticalness,
- periodic execution,
- dependency on I/O,
- functional cohesion.

Better response time is achieved for time critical activities by using high priority processes. Activities driven by the clock interrupt or I/O are also placed into separate processes so that interrupt servicing may proceed concurrently with other activities. Many of the services of the station operation group do not fall under any of the first three criteria. They could have all been realized using a single process. Nonetheless, they have been separated into a small number of distinct processes which provide relatively independent services. Separate processes are advantageous since a software error in a service can result in suspension of the offensive process. The station may then be able to continue operating with other services unaffected. This is particularly helpful during kernel development, and also improves the reliability of the station software in general. Each process has sufficient functional cohesion that there is negligible overhead introduced by communication between service processes.

For decomposition of the distributed messages system, the main consideration was dependency on I/O. A separate process is used to service each link type. At present, only serial links are used, but different types of link may be added in the future.

3.6.2.2 Processes Priorities

Priority has been used to influence response time of kernel processes. If a process of higher priority than the currently executing process becomes ready, then it will preempt the executing process. The higher priority processes at a station should only perform a small amount of processing each time they become ready. This avoids degrading the response time of other high priority processes, which could prevent them from correctly carrying out their functions.

Range[*]	**Processes**
a	Exception Handler
b	Timer Console Log
c	Serial LDD/IODD
d	Router
e	Process Manager Profiler S Record Loader Operator Interface Connection Database Debugger
f	Init

[*] In order of highest to lowest priority, with range *a* being the highest.

Table 5 Relative priorities of kernel processes

Table 5 shows the relative priorities of the kernel processes. This arrangement may be adjusted as more experience with NetKernel is gained. User processes may use the same priority ranges as kernel processes except for the highest, range *a*. Additional priority levels between the ranges of Table 5 are also available for user processes.

Exception Handler is at the highest priority so that it can service exceptions generated by any other process, particularly error exceptions. Group *b* is for critical kernel processes. It is particularly undesirable to delay the processing of messages by Timer. Any such delay in processing of a timer request is added to the interval which Timer measures, reducing the accuracy of the Timer service. Console Log is also placed at high priority to avoid delaying of log messages, which are used to report critical events. So although user processes may be placed at priority higher than range *b*, this should be limited to device drivers requiring very fast response times.

Ranges *c* and *d* are used for the processes of the distributed message system which actively participate in the propagation of messages throughout the physical network. Serial LDD/IOD has been placed at the higher level since it also functions as an I/O device driver. Some I/O devices may not support flow control so a fast response time is desirable to avoid the loss of characters. The placement of user processes with respect to these priorities depends on the application. User processes at priorities immediately above *c* or *d* have the potential to reduce the performance of the distributed message system. This can be an appropriate design decision since local activities often require a faster response than distributed activities. The performance of local messaging is unaffected since it is handled entirely by the microkernel.

Range *e* is used for non real-time kernel processes. These processes provide management services which should not interfere with the real-time operation of user processes. Hence, the priority of real-time user processes should be greater than *e*. The Init process runs at the very lowest priority. It executes an empty loop whenever there is no other ready process at a station.

3.6.3 Station Operation

Each of the processes in the Station Operation group shall be described. Treatment of the simpler processes will be quite brief. The client-server and data flow process models were presented in Section 2.3.5. All of the processes in this group except for Init and Default Exception Handler act as servers, accepting requests for a service and returning reply messages. Serial LDD/IODD also acts as both a server and a transform process. In the latter role, it exchanges data between processes and the serial port. Default Exception Handler and Init have supervisory roles in the operation of the kernel.

The processes of this group are fairly independent of each other. Although the Management Services make use of Basic Services, the latter were designed to provide generally useful services rather than to cater to the needs of any particular Management Service. Process Manager, Default Exception Handler, and Debugger are all related in that they directly control the processes of a station. All three of these processes access the PCBs of processes, as well as other kernel data structures related to their specific roles. These three processes shall be presented first, followed by the remaining Management Service processes and then the remaining Basic Service processes.

3.6.3.1 Process Manager

The Process Manager provides services for process management. These include commands for setting up, starting, suspending, resuming, and killing processes. Information such as a list of processes or the status of an individual process can also be requested from Process Manager. As an example of the message syntax for a kernel process, the command and reply formats for Process Manager have been included in Appendix B. The operations which Process Manager performs in response to several of its more important commands will now be considered.

The Allocate command to Process Manager creates a new process at a station. An identity is established for the process and some resources are reserved for it. The identity consists of a process id number, referred to as a *pid*. Process Manager allocates and initializes a PCB for the process. The microkernel and other kernel processes will access the PCB at later times for purposes such as scheduling and recording statistics, but only Process Manager creates and destroys PCBs. The control state of the new process in its PCB is initialized to New (Section 3.5.3.1), so the process is not yet eligible to be scheduled. This permits other setup activities to be performed prior to starting the process.

The context (i.e. stack pointer and other registers) of a nonexecuting process is stored in its PCB. Commands to Process Manager are used to set up the context of a process prior to starting it. Memory regions are also reserved for a process prior to loading it. All processes execute in a common address space. Process manager allows up to five non-contiguous regions to be reserved in this space for each process. Process Manager does not provide any allocation scheme for regions. That is considered the responsibility of a higher level entity such as a development tool which manages processes from a workstation (i.e. HIP). Requests to reserve a region for a process include the start address, end address, and a usage tag: private or sharable. The latter facilitates code and data sharing between processes. Requests either succeed or fail depending on whether the request conflicts with previously reserved regions. Regions are intended to satisfy long term requirements of a process for large sections of memory, such as code, data and stack. Smaller requirements may be satisfied using the system calls for dynamic memory allocation.

The outlets of a process are typically connected to input queues prior to starting the process. It is also possible for a process to establish connections on its own and for connections to be changed while a process is executing. In all cases, an outlet is connected to an input queue by sending a Connect command to Process Manager. Process Manager makes use of the services offered by Connection Database which maintains Connection Blocks (CB) for all connected outlets.

Upon receiving a Start command for a process in the New state, Process Manager changes its states to Enabled and Ready and places it at the end of the process queue for its priority. Process Manager also accesses the process queues to service the Suspend, Continue, and Kill commands. The Kill command results in freeing of all kernel resources which were allocated to the process being killed, including its PCB, dynamic memory blocks, and timers.

The Process Manager also accepts commands for associating exceptions with handler processes. It directly accesses the ECBs to record the pid associated with an exception.

To summarize, the Process Manager performs various operations associated with process management. It allocates, initializes, and deallocates PCBs as processes are created and killed. It changes the state of processes and removes/adds them to process queues as they are started, suspended, continued, and killed. Process Manager dialogues with Connection

Database to establish CBs for the connection of outlets. Process Manager also accesses ECBs to associate handlers with exceptions. Some additional minor functions are provided which have not been mentioned.

3.6.3.2 Default Exception Handler

Default Exception Handler (DEH) is the default handler for most of the M68000 exceptions. Special exceptions are listed in Section 3.5.6, "Exception Catcher". Kernel initialization and user activities may select different handlers for some exceptions such as device interrupts, but typically DEH remains the handler for many exceptions. When activated, DEH takes response actions based on the particular exception and circumstances.

DEH interacts with the Debugger process. When the execution of a some process produces an exception handled by DEH, DEH responds by suspending the process and sending a message to Debugger. This enables the Debugger to provide tracing, breakpoints, and generation of status messages for unexpected exceptions such as a bus error or a divide by zero.

There are several unrecoverable errors which cause Default Exception Handler to shut down its station. Upon shutdown, the execution of processes is abandoned until a reset is performed. Diagnostic information such as register contents at the time of the error is recorded into the Diagnostic Block (DB). A low-level monitor is then activated which supports the dumping of memory contents for postmortem analysis. Unrecoverable errors include unexpected exceptions such as bus error during the execution of the microkernel or a handler process (bus error during the execution of a user process simply causes that process to be suspended). The occurrence of an unexpected interrupt is also an unrecoverable error.

3.6.3.3 Debugger

Debugger provides facilities for low-level debugging of processes. Debugging at the level of machine code is supported. Additionally, various NetKernel variables and statistics concerning a process may be examined. Debugger provides commands for tracing, breakpoints, examining/modifying registers and memory, and suspending/resuming processes. Breakpoints and tracing involve interaction with Default Exception Handler (DEH), as described earlier. Most of the debugging functionality is contained in Debugger, with DEH simply notifying Debugger of exceptions. The functionality contained in the handler is minimized to simplify development as recommended in Section 3.5.6.3.

Debugger has read access to most of the kernel data structures in order to allow display of various kernel variables. Debugger has read/write access to PCBs to allow examining and modifying of process context and state. Some of the process control commands of Debugger that overlap with those of Process Manager are implemented by sending messages to Process Manager.

3.6.3.4 Serial LDD/IODD (IODD role)

Serial LDD/IODD manages the serial ports of a station. More specifically, it controls the DUART peripheral chips associated with the serial ports. The DUARTs have a memory mapped register interface which supports asynchronous character-oriented serial I/O. Since it is a device driver, Serial LDD/IODD is implemented as a handler process.

Serial LDD/IODD provides two modes of operation for each port of a station: link mode and I/O mode. The modes of different serial ports are independent and may be changed dynamically at runtime. For instance, it is possible for some ports to be in link mode while others are in I/O mode. The two modes correspond to the Link Device Driver (LDD) and I/O Device Driver (IODD) roles of Serial LDD/IODD, respectively. The justification for supporting the two roles with a single process was presented in Section 3.4.3. This subsection describes how the mode of a port is switched and the IODD role.

At start-up all serial ports default to link mode. A port may be changed from link mode to I/O mode using either of two methods. The first is to send a control message to Serial LDD/IODD. The second method allows the I/O device itself to initiate I/O mode on its port. This allows an operator to switch a port to I/O mode from a terminal attached to the port. The operator waits for a timeout period to elapse, and then types a special character at the terminal. The special character is necessarily different from the first character of a link layer packet so that Serial LDD/IODD can recognize it as a request to switch from link mode to I/O mode. The same two methods, control message or special character, are available to switch a port back to link mode.

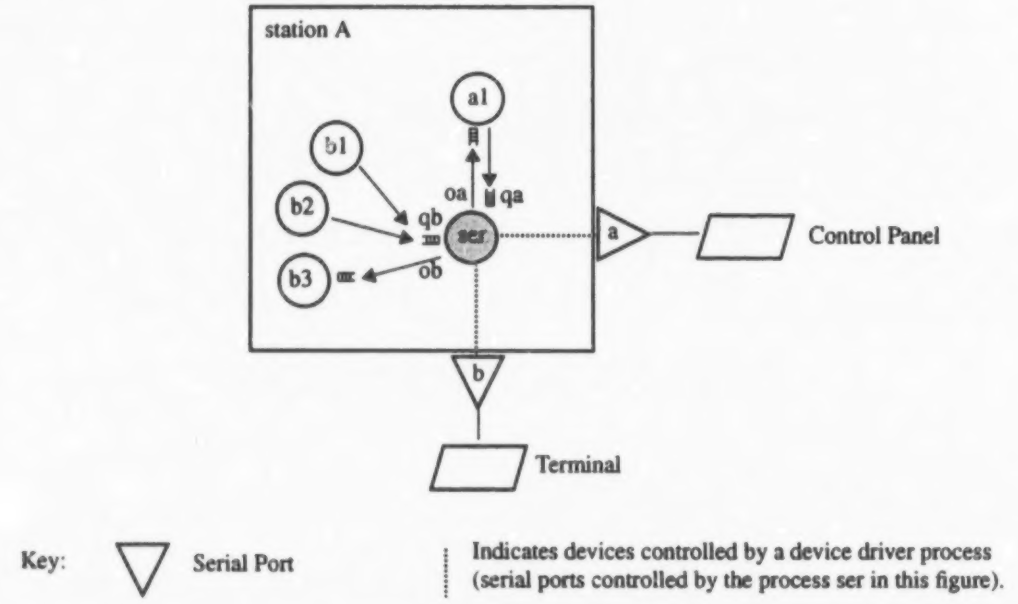

station A
a1
b1
b2
b3
oa
qa
qb
ob
ser
a
Control Panel
b
Terminal
Key:
Serial Port
Indicates devices controlled by a device driver process (serial ports controlled by the process ser in this figure).

Figure 19 Example of Serial LDD/IODD managing two I/O devices.

Serial LDD/IODD supports message-oriented I/O with devices attached to ports which are in I/O mode. Associated with each serial port is an outlet and an input queue of Serial LDD/IODD. This is illustrated in the simplified example of Figure 19. The Serial LDD/IODD process (*ser*) at station A is managing two serial ports (*a* and *b*), both of which are connected to I/O devices. Process *a1* sends output to Control Panel by sending messages to queue q*a* of *ser*. Process *a1* also receives input from Control Panel as messages sent from *ser* through outlet *oa*. Three processes interact with Terminal. Processes *b1* and *b2* send output to the terminal. Process *b3* receives input from the Terminal. The messages from *b1* and *b2* to the terminal will be interleaved—they are outputted to Terminal in the order that they arrive at queue *qb*. To avoid interleaving of messages, a locking mechanism between *b1* and *b2* could be provided at the user level. Note that a Serial LDD/IODD process can manage more than two ports, which would involve more queues and outlets than shown in the Figure. Also, the queue of *ser* to which control messages are sent for switching the mode of ports is not shown.

Characters received from a device by Serial LDD/IODD are accumulated into an input message until some end-of-message character (such as carriage return or line feed) is received from the device. Serial LDD/IODD then sends the input message through its outlet which is associated with the device. Each output message for a device received by Serial LDD/IODD is simply transmitted one character at a time to the device.

3.6.3.5 Operator Interface

Operator Interface provides a command line interface for dialoguing with processes. It is used in conjunction with another interface process, that handles input/output, such as Serial LDD/IODD. The interface process reads and displays lines to a device such as a terminal or a workstation window.

Operator Interface forwards command lines typed by the operator to a *destination* process. Any reply messages from the destination are forwarded back to the operator. The destination is selected using control commands to Operator Interface which begin with a special escape character. Control commands are interpreted by Operator Interface rather than being forwarded to the destination process.

Figure 20 illustrates how dialogue with processes is performed from a terminal using Serial LDD/IODD (*ser*) and Operator Interface (*opi*). Lines typed at the terminal are collected by *ser* and sent as messages to *opi*. *Opi* filters out control messages, which it interprets to select a destination process (such as *x* or *y*). Destination processes may be local or nonlocal. The selection of a new destination process causes *opi* to send a message to Process Manager (the connection from *opi* to Process Manager is not shown in the figure) to change the connection of outlet *oa'* to the new destination process.

Operator Interface supports multiple sessions, with each session having an independent destination process. Each session sends its messages to a different input queue of Operator Interface. Only one session is shown in Figure 20. Two sessions, for instance, could be

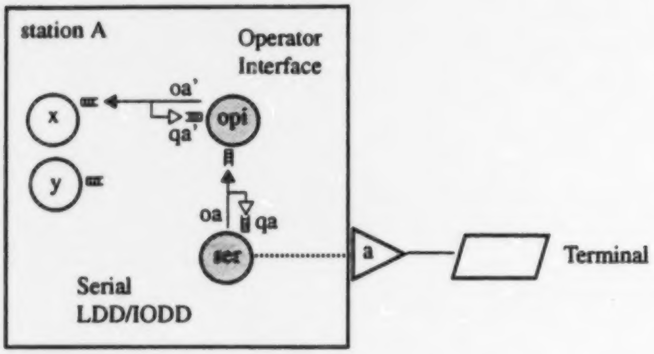


Figure 20 Dialogue between a terminal and destination *x* through Operator Interface

used to allow two terminals attached to a station to perform independent dialogues with different destination processes. Extending Figure 20, different input queues and outlets of *ser* and *opi* would be associated with the second terminal.

3.6.3.6 Profiler

Profiler reports profiling information maintained by the kernel. Commands to profiler may request a single report for one or more processes, or may request the periodic generation of reports. Periodic reports allow the resource usage of processes to be tracked over time. Profiler makes use of a periodic timer from the Timer service to provide periodic reports. Presently, statistical cpu usage (Section 3.5.7.1) and dynamic memory usage are available in Profiler reports.

3.6.3.7 S Record Loader

S Record Loader is used to load object code to a station. It accepts messages containing Motorola S records to be loaded into memory. A few other commands provide simple functions such as setting a relocation offset, reporting the number of S records received, etc.

3.6.3.8 Timer

Timer provides three types of timer services: one-shot, periodic, and shared. Timer accepts command messages for setting up, resetting, and cancelling timers. A process, referred to as the owner, may request a new timer by sending a setup message to Timer which specifies the timer type and an interval. Upon expiry of the interval, Timer sends a timeout message back to the process.

Shared timers are a service included specifically to reduce the overhead associated with handler processes (Section 3.5.6.3). When a shared timer is set up, Timer returns a memory address to the owner. The current value of the timer is stored at this address. The owner is allowed to change the value directly. This avoids the overhead of having to send messages to Timer in order to change a timer value. For serial LDD/IODD, which needs to reset the timer associated with a link each time a new character arrives, the overhead saved is substantial.

Timer manages a Timer Control Block (TCB) for each active timer. As discussed in Section 3.5.7.3, the microkernel decrements the countdown values of timers and sends a message to Timer whenever there is a timeout. This reduces the overhead incurred by Timer. Upon receiving a message from the microkernel, Timer sends a timeout message to the owner of the timer.

3.6.3.9 Log

Critical status messages are sent to the log process. These are recorded locally and forwarded to a central output device. The local record is accessible during regular operation, postmortem diagnosis, and following a reset. Two buffers are used for the log, with one always containing log messages which have been preserved from prior to the most recent reset. A central output device can be specified following start-up by connecting an outlet of the Log process. By connecting all log processes to some central process such as a device driver for a printer, log messages from all stations can be displayed at a single device.

3.6.3.10 Init

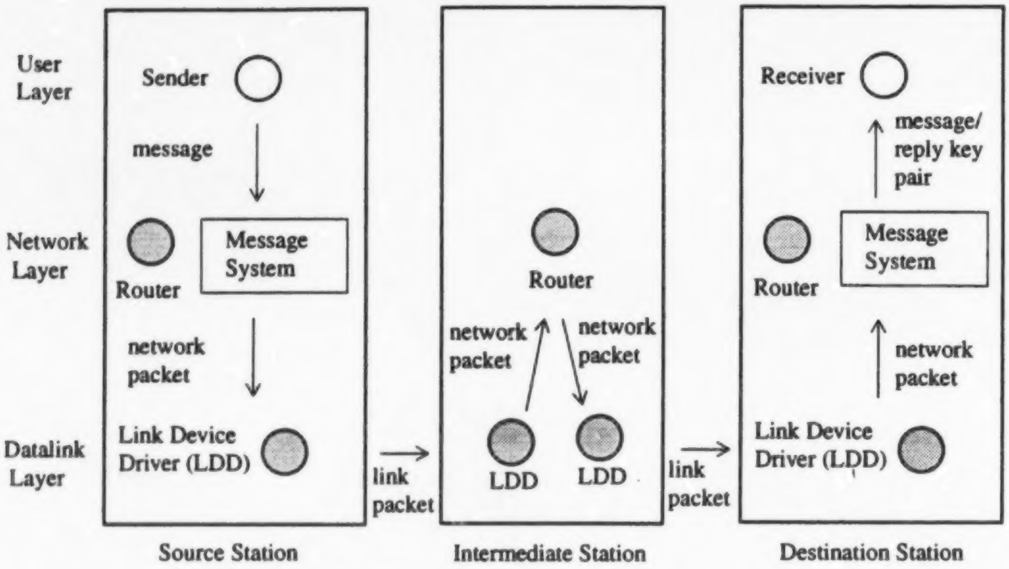
At reset, two processes are bootstrapped by start-up code: Process Manager and Init. Init makes use of Process Manager's services to set up and start the remaining kernel processes. Both the start-up code and Init also initialize various kernel data structures. After initialization is completed, Init executes an empty infinite loop. Init is at the lowest priority possible, so the empty loop is executed whenever no other processes at a station are ready to run.

3.6.4 Distributed Message System

3.6.4.1 Overview

The organization of the distributed message system into a data link layer and a network layer has already been described. Figure 21 illustrates the data path of a nonlocal message through these layers. The figure also shows the NetKernel entities at each step and the data units exchanged between layers. The case of a single intermediate station is illustrated, but other cases are similar.

The data path begins when a process performs a Send() or Reply() to pass a message to a nonlocal process. The microkernel module Message System handles the system call. For Send(), it determines that the message is nonlocal by examining the Connection Block associated with the outlet through which the message is being sent. The information in Connection Blocks is maintained by the Connection Database process. The Process Man-



network packet = network header + message
link packet = link header + network packet + link trailer
key = pointer to network header

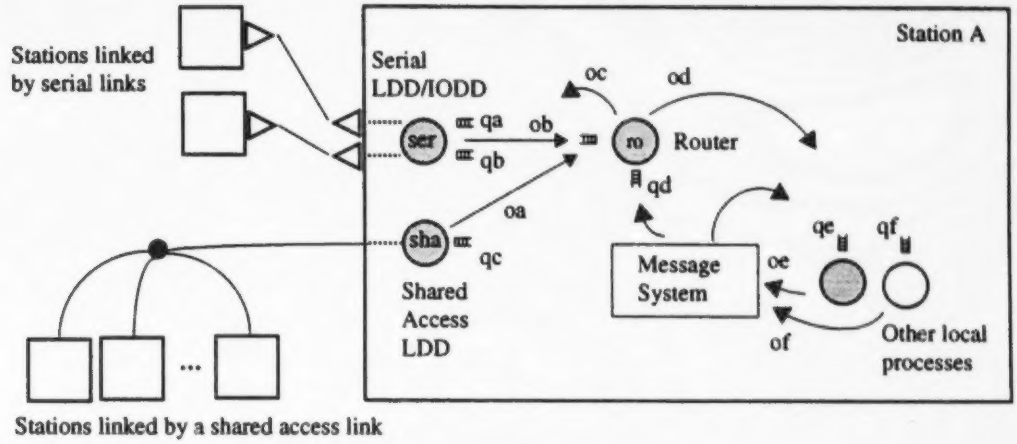
Figure 21 Data path and units exchanged for a nonlocal message

ager makes use of Connection Database services to establish connections between processes. For Reply(), the Message System determines that the message is nonlocal by examining the header to which the reply key points.

Having determined that the destination is nonlocal, the microkernel generates a network packet containing a header followed by the message. The header contains items L1 through L5 (Section 3.5.5.1), including the route which the packet will take through the network. The Message System module then passes the Network Packet to Router.

Router examines the header to determine the link over which the message should be transmitted for its first hop through the network. It then passes the network packet to the Link Device Driver (LDD) that manages the link. The LDD generates a link packet suitable for transmission over the link. This may involve the addition of frame delineating codes, length information, error detection codes, etc. The link packet is then transmitted over the link to the second station along the route. The receiving LDD at that station unpackages the network packet and passes it up to Router. Router examines the header and determines the LDD to be used for the packet's second hop through the network. Router passes the network packet down to that LDD for transmission to the third station along the route.

Router at the third station determines that the network packet has finally arrived at its destination station. It passes the message and reply key to the destination process (receiver), which is local to that station.



Summary of Outlets:
oa, ob: LDDs send arrived network packets to Router.
oc: Router sends outgoing network packets to LDDs qa, qb, or qc, depending on outgoing link.
od: Router deposits arrived messages (with reply keys) into input queues of local destination processes.
oe, of: Local processes performing nonlocal sends. A local send causes Message System to deposit a message/key pair into the input queue of the local destination process. A nonlocal send causes Message System to send a network packet to Router.

Figure 22 Distributed message system processes at an example station

Figure 22 shows the elements of the distributed message system from a different perspective. The connections between processes at a single hypothetical station are shown. The station is directly linked to two other stations through two serial ports. It is also linked to a group of stations through a shared access link. Hence two LDDs are active at the station: one to manage the serial ports and one to manage the shared access link. Router and the LDDs direct the flow of nonlocal messages from source to destination, forming a data flow style of software network.

An LDD has a separate input queue for each physical link that it manages. Network Packets for each link are deposited by Router into the corresponding input queue. The LDD may service its queues concurrently since messages may be transmitted over different links at the same time. The LDD *ser* in the figure manages two serial links. Hence it has two input queues into which Router deposits network packets. The LDD *sha* manages a single shared access link. Hence it has only a single input queue into which Router deposits its network packets.

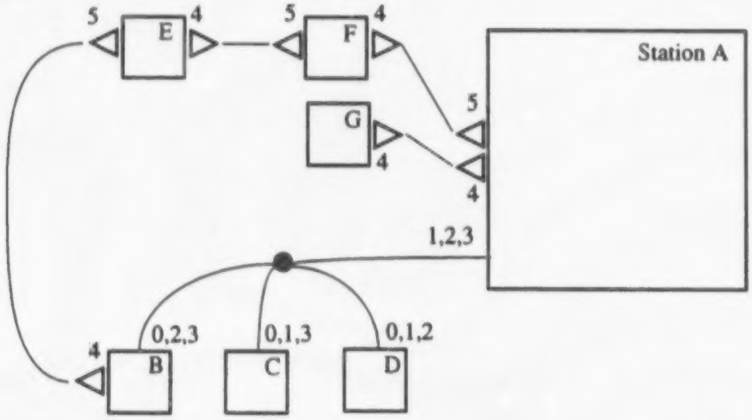
3.6.4.2 Connection Database

The Connection Database process manages Connection Blocks. A unique connection block is associated with each connected outlet of a process. The block records the destination pid, input queue, and route to be used. If the destination process is local, then the route is null. Presently, Connection Database does not provide enough functionality to justify a separate process. It was provided as a stub which may be expanded in the future to explore dynamic routing. Since Connection Blocks contain routes, this process is in a good position to change routes in response to network conditions such as congestion. Currently, Connection Database simply provides a service for use in setting up Connection Blocks. This service is intended for use only by Process Manager (to establish connections between processes).

3.6.4.3 Router

Router receives network packets of local origin from Message System and incoming network packets from the LDDs of a station. It performs a small amount of processing on the header of the packet. It then either passes the packet to an LDD for transmission to another station, or passes the message/key pair of the packet to a local process. The latter occurs once a network packet has arrived at its final destination station.

Router determines whether or not the network packet has arrived at its final destination by examining the route length and index contained in the header. The length gives the number of hops to be taken through the network. The index indicates how many hops have been completed. Router increments this index as a message propagates through the network.



A directly to F: <5>
A to F through stations B and E: <1, 4, 4>

A to E through station B <1,4>
E to A through station B <5, 0>

Figure 23 Some routes through a hypothetical physical network.

The routing fields used in Connection Blocks and network packet headers contain a sequence of *Virtual Link Numbers* (VLN) which indicate the link to be taken for each hop. The VLN for each hop has meaning only with respect to the station at which the hop originates. It uniquely identifies one of the directly linked stations as the destination of the hop. In Figure 23, the VLNs recognized by each station have been shown next to the corresponding links of a hypothetical physical network. Station A, for instance, recognizes VLNs 1 through 5. Stations E recognizes only VLNs 4 and 5. The assignment of VLNs is arbitrary, but it can be helpful to adopt conventions. For instance, on a station with two serial ports, we are currently using the convention that VLNs 4 and 5 are used to refer to the ports physically located on the left and right sides of the expansion card, respectively. In Figure 23, we have also adopted the convention that VLNs 0 through 3 are used to refer to stations A through D, respectively. Each of these stations recognizes only three of the four VLNs from 0 through 3 since it would not make sense for it to transmit a message to itself over the link. Note that there is no requirement for VLNs to be unique in any way throughout the network. VLNs must only be unique at a station.

A knowledge of the association between VLNs and links is required to specify the route for connections between processes at different stations. The route is included as a parameter to the Connect command of Process Manager. Some example routes are shown in Figure 23. Note that the reverse of a route is not necessarily equal to the reverse sequence of VLNs (see the routes for A to E and E to A). The Reply() system call makes use of the reverse route (Section 3.5.5.4). The sequence of VLNs for the reverse route is recorded by Router in the header of a packet as it propagates through the network. The reverse route is then available should the destination process send a reply message.

Router makes use of an internal *VLN-to-LDD Translation Table* (VLTT). It describes the association between VLNs and links by associating each VLN with a LDD. Our current implementation generates this table at start-up by testing for the presence of links and using some simple rules to generate VLNs. The table maps each VLN to a 3-tuple of the form ⟨*LDDpid*, *LDDqid*, *SharedID*⟩. *LDDpid* is the pid of the LDD which manages the link associated with the VLN. *LDDqid* is the input queue of the LDD into which network packets for the link should be deposited. *SharedID* is used only for shared access links—it provides an id that may be used by the LDD to identify the destination station on the link. This is required since there may be many stations attached to a shared access link. Router sends a network packet on its next hop by looking up the VLN of the hop in VLTT. It sends the packet to the indicated LDD, passing sharedID along as well.

3.6.4.4 Serial LDD/IODD (LDD role)

Serial LDD/IODD manages the serial ports of a station. It serves two roles: link device driver (LDD) and I/O device driver (IODD). This subsection describes its role as an LDD. The IODD role and switching between modes were described in Section 3.6.3.4. All serial ports default to link mode upon reset, making them immediately ready for exchange of link packets between stations.

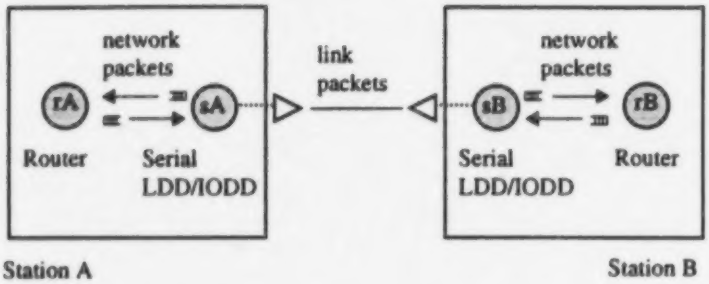


Figure 24 Communicating Serial LDD/IODD processes at adjacent stations

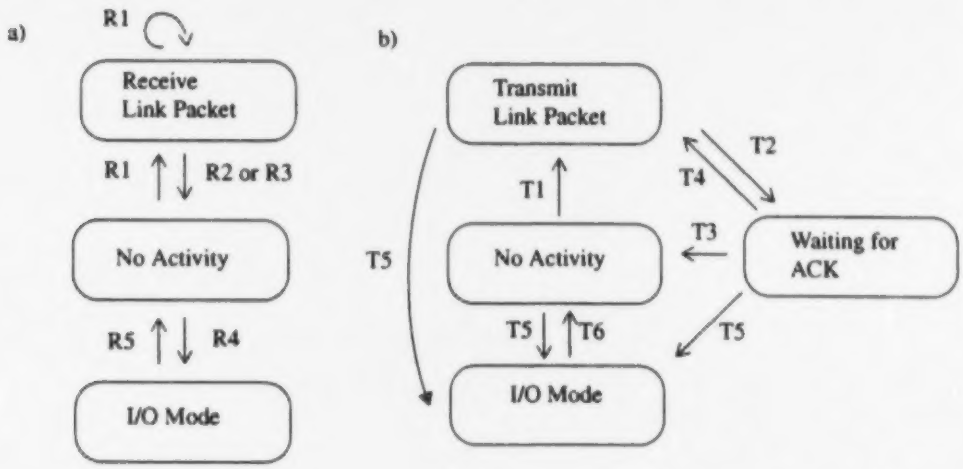
Serial LDD/IODD transmits and receives outgoing and incoming link packets on serial ports that are connected to other stations. Figure 24 illustrates the role of Serial LDD/IODD in a simple physical network of two stations connected by a serial link. Consider the transmission of a message from station A to station B. The Serial LDD/IODD processes (*sA*, *sB*) and router processes (*rA*, *rB*) of both stations are all involved. Router *rA* passes the outgoing network packet to *sA* for transmission. A header and trailer are added by *sA* to form a link packet which is then transmitted over the serial port. After receiving the incoming link packet, *sB* removes the header and trailer. The resulting network packet is passed to Router *rB*.

Consistent with the design philosophy of Section 3.2, a fairly simple communication protocol has been adopted for Serial LDD/IODD: the one bit sliding window protocol. This protocol is described in [Tanenbaum90, pp. 225-228]. A full duplex version with backward error recovery is used here. It was selected because it is one of the simplest protocols that is robust. It is inefficient under certain circumstances, which may motivate the use of a more sophisticated protocol in the future. Tanenbaum describes its robustness: "No combination of lost frames or premature timeouts can cause the protocol to deliver duplicate packets to either network layer, or to skip a packet, or to get into a deadlock" [Tanenbaum90, p. 226]. Checksums are used to detect errors in arrived link packets. The occurrence of an error results in retransmission of the packet.

This protocol is inefficient in the presence of errors since an error results in the link going idle in one direction until a timeout occurs. This inefficiency may be addressed by extending the protocol to support negative acknowledgment (NACK), but this has not been done yet. The protocol's small window size also makes it inefficient when the propagation delay of the link approaches or exceeds the transit time of a packet. Since Serial LDD/IODD manages relatively low bandwidth links over short distances, the window size of one does not impact link level performance in our design.

The DUART hardware managed by Serial LDD/IODD provides asynchronous character-oriented serial I/O with hardware flow control. A benefit of hardware flow control is that a slow response time in servicing incoming characters does not cause characters to be lost.

This is important since a station will occasionally execute higher priority processes than Serial LDD/IODD. Above the datalink layer, NetKernel provides no flow or congestion control (Section 3.4.2).



R1. START arrives
R2. END arrives
R3. Timeout occurs
R4. Pending request to switch port to I/O mode
R5. Pending request to switch port to link mode

T1. Network packet received from Router
T2. Finished transmitting START, link packet, and END
T3. ACK with correct sequence number arrives
T4. Timeout occurs
T5. Pending request to switch port to I/O mode
T6. Pending request to switch port to link mode

Figure 25 State Diagrams for (a) receiver and (b) transmitter

Conceptually, a state machine is associated with each of the receivers and transmitters managed by Serial LDD/IODD. Figure 25 shows the two state diagrams. Most of the transitions in these diagrams occur due to the arrival of a control character or the occurrence of a timeout. The following control characters are used in the protocol.

- START: Marks start of link packet
- END: Marks end of link packet
- ACK: Positive Acknowledgment of successfully received linked packet. A single bit in this code indicates the sequence number of the packet being acknowledged.

3.7 Correctness of Design

No rigorous or formal attempts have been made to completely prove the correctness of the kernel design. Analytical proof methods for concurrent processes are only just emerging (eg. Communicating Sequential Processes, [Hoare85]). This section informally considers two logical properties required for the correctness of concurrent software: *safety* and *live-*

ness. An additional physical property, *timeliness*, becomes of interest for systems with hard real-time requirements [Stankovic88]. Real-time properties will not be considered here. Mapping techniques to meet real-time requirements are an important area for future research.

The correctness of a sequential program can be shown by proving that it terminates with the correct result. Demonstrating correctness for kernel software must be done differently than for ordinary sequential programs.

> ...the single most distinguishing feature of an operating system or real-time system is that it must never halt. The only way to halt a typical operating system is to push the start button on the computer panel. An operating system prints nothing of its own (except some non-essential logging and accounting data). Thus when studying operating systems, we must revamp our notions of what it means for a program to be correct... In our abstraction, we shall distinguish two types of correctness properties: safety properties and liveness properties [Ben-Ari82].

We shall use the concepts of safety and liveness as defined by Ben-Ari.

The analysis of this section considers only the correctness of an individual station. This is appropriate for evaluating the kernel since 1) it is replicated at each station, and 2) stations do not communicate until user software networks are loaded. For user software networks which are distributed among stations, it would be necessary to consider correctness at both the level of individual stations and the level of the entire network.

3.7.1 Safety

Safety properties are static requirements which must be satisfied throughout the execution of concurrent software. Mutual exclusion is the only safety property which will be considered. A shared data structure must not be accessed concurrently by the microkernel and one or more kernel processes such that inconsistent data is read or written. This has been achieved by blanketing code which accesses shared data structures in critical sections, as described in Section 3.5.2.2 and Section 3.6.1.

3.7.2 Liveness

Liveness properties are dynamic requirements reflecting events which should occur under various circumstances during the execution of concurrent software. "Liveness means that if something is supposed to happen then eventually it will happen" [Ben-Ari82]. The following three subsections consider the liveness issues of deadlock, lockout, and termination of system calls.

3.7.2.1 Deadlock

Deadlock is the most dramatic violation of liveness.

> A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause [Tanenbaum92, p. 242].

Circular arrangements of client-server processes can result in deadlock. For instance, consider process *A* which sends a message to *B* and then waits for a response. If *B* then sends a request message (but not the reply for which *A* is waiting) to *A* and waits for a response, then the two processes will be deadlocked. Processes *A* and *B* are each waiting for the other to respond. The potential for this type of deadlock can be detected by drawing a dependence graph of client-server relationships. An arc from process *A* to *B* indicates that *A* acts as a client with respect to *B*, making use of a service offered by *B*. A sufficient condition for the absence of deadlocks due to circular client-server relationships is that the graph contain no directed cycles.

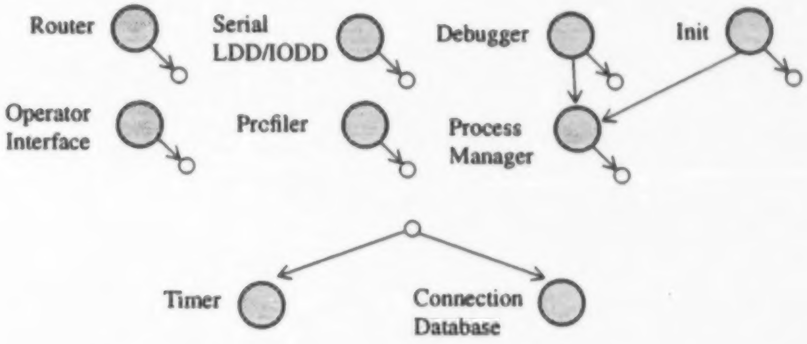


Figure 26 Client-server relationships among kernel processes

Figure 26 shows the graph for kernel processes which operate in client or server roles. Only client-server relationships, not data flow relationships, are shown. Kernel processes which do not participate in any client-server relationships have been omitted. To simplify the graph, several processes that operate exclusively in server roles have been shown at the bottom. Since these processes make no use of other services, it is not possible for them to be part of a cycle. The graph is simplified considerably by not drawing all the individual arcs to these processes since their services are used extensively by the other processes. The lack of cycles in Figure 26 shows that the kernel processes do not deadlock due to circular client-server relationships. Similar analysis may be performed for user processes, to ensure that their addition does not introduce the possibility of deadlock.

3.7.2.2 Lockout

Lockout, also referred to as starvation or livelock, occurs when one or more processes are indefinitely delayed from obtaining a resource due to the activities of other processes. The kernel manages two resources: the cpu and the links of a station. Processes acquire the cpu for execution. Links are accessed indirectly by sending messages, with the sender being unaware that it is using the link resource.

We first consider the potential for CPU and link lockout under NetKernel. Next, avoidance of lockout is considered.

CPU Lockout

The microkernel implements the following simple scheduling policy: The highest priority ready process gets the cpu. Lockout is possible under this scheme. If one or more high priority processes take up all the cpu time, then a lower priority process will never get to execute.

Link Lockout

Packets to be transmitted over a particular link are serviced on a First-Come-First-Serve basis using queues. Any packet placed in the queue for a link will eventually be serviced, so lockout does not occur. Due to the finite length of the input queues, however, it is possible for packets to be lost due to overflow. This occurs when packets are deposited into a queue faster than they can be transmitted. This is not technically the same as lockout, since there is no discrimination between processes. All processes are equally likely to lose packets going to a full queue. But the end result is similar to lockout: access to a resource is lost.

Avoidance

When the kernel executes by itself, there is little demand on the cpu and no demand on the links. When user processes are added, however, CPU lockout and packet loss become possible. Avoidance of these problems for embedded system software and purely computational software will each be considered separately.

In an embedded system, data flow organization of processes is common. The system must keep up with incoming data by performing all necessary transform and output operations. Simple conditions which need to be satisfied are suggested by the preceding discussion of lockout:

N1 (sum of cpu demands of processes at station) ≤ (cpu capacity of station),
N2 (sum of demands on link) ≤ (link capacity)

Evaluating these conditions can be difficult in practice because it is hard to quantify the demands of processes. One approach is to make conservative estimates which are satisfied by mapping and validated by testing. Providing general guarantees is not possible with this approach. Rather we rely on our understanding of the program's demands and the hardware's capacity to ensure correct operation. This is less than ideal. It would be preferable if logical analysis alone could guarantee correctness.

Purely computational or off-line programs are not driven by a real-time environment. These programs can be written to avoid CPU lockout and packet loss. Avoidance of CPU lockout requires that a program is specially designed to be fair. Arbitration between competing processes must be programmed at the user level, since the NetKernel priority scheme is inherently unfair. Packet loss can be avoided by limiting the number of messages which are allowed into the network at once. As a simplistic example, consider a system where all the queues are length 10. If the software network contains 5 user processes, each having a maximum of two outstanding messages, then none of the queues will ever overflow. Flow control between processes could be used to limit the number of messages a process injects into the network. This is essentially a very simple scheme to avoid packet loss from congestion, with many more sophisticated schemes known.

To review our consideration of lockout, the kernel by itself makes minimal demands on resources and there is no possibility of lockout. The presence of a user software network increases the demand for resources, making cpu lockout and packet loss possible. For embedded systems, these can be avoided by considering performance issues. For off-line software, user level mechanisms provable by logical analysis can avoid the same problems. In practice, software may have both embedded and off-line components, making the application of these techniques even less straightforward.

3.7.2.3 Termination of System Calls

Wait() is the only system call which is allowed to block the caller in this design. All system calls must return control to some process within a finite time. This requirement upon system calls is a liveness property. The absences of infinite recursion among the system calls is a necessary condition for a call to return. An approach similar to the earlier dependence graph (Figure 26) was taken to verify this condition. A structure chart of the routines in the microkernel (not presented here) was drawn. A structure chart shows each routine as a node. Routines which directly call each other are connected by a directed arc from the caller to the callee. The presence of cycles in the chart indicates the potential for infinite recursion.

One problem in an early design iteration was uncovered through the structure chart analysis. When an attempt to allocate memory by the microkernel failed due to lack of free memory, the microkernel would call Send() to pass a warning message to the Log process. Send(), however, needs to allocate memory as part of its operation. The design was refined to detect this case and shut down the station in an orderly fashion. The orderly shutdown includes recording the reason for failure as opposed to the previous behavior where recursion would simply continue until stack space ran out. Other than the case just noted, there were no other cycles in the structure chart. Hence we can be confident that no system calls fail to terminate due to infinite recursion. Other errors, such as an infinite loop, could still result in a system call which fails to terminate. Nonetheless, one non-trivial source of design errors has been eliminated.

To summarize the preceding sections treating correctness, it has been informally demonstrated that the high-level design of the kernel satisfies various aspects of liveness and safety. The treatment is far from a complete proof of these properties, but it does improve our confidence in the design.

3.8 Implementation

This section presents a few aspects of the NetKernel implementation on the M68000 target stations. Tools, source code statistics, and several refinements of the design are presented. The final subsection briefly discusses the NeXT workstation prototype of NetKernel.

3.8.1 Compiler and Other Support Software

NetKernel was developed using the CrossCode C package [Software88]. This package includes a C compiler, M68000 assembler, linker and several other miscellaneous tools. It is well suited to the development of embedded ROM applications, providing good control over the placement of program segments in the memory space of the target system. A cross-development version which runs on the department's HP9000 computer system was used.

Debugging at the target stations was performed using monitor/debug firmware from Motorola called TUTOR. TUTOR supports debugging at the assembler level, providing functions such as instruction disassembly, register examine and modify, breakpoints and tracing.

3.8.2 Code Statistics

Some source code statistics for NetKernel are included in Table 6. The source contains about 7500 lines, which excludes comments, blank lines, and .h header files. File types .s and .c contain assembler source code and C source code, respectively. Most of the code is written in C for reasons of maintainability, portability, and ease of development. Some sections were written in assembler for either efficiency or direct access to the cpu architecture. For instance, the microkernel code for context switching and clock servicing was written in assembler. Substantial portions of Serial LDD/IODD were written in assembler to minimize the overhead of passing link packets through the serial ports.

Section of Kernel	**Lines of code**		
	.s files	.c files	Total
Microkernel, including system calls	708	1533	2241
Kernel processes	1219	4155	5374
Total	1927	5688	**7615**

Table 6 Source code statistics

3.8.3 Process Priorities

Software priorities numbered from 0 to 14 have been adopted. The priority of kernel processes have been assigned within this range to preserve the relative priorities presented earlier in Table 5. The M68000 interrupt mask levels of 0,1,2,...,7 correspond to software priorities of 0,2,4,...14, respectively. Several unused ranges are present at levels appropriate for different classes of user processes, such as device drivers and low priority processes. User processes may also be placed at the same priority level as kernel processes other than Exception Handler.

3.8.4 Developing User Processes

User processes are written as sequential programs and compiled with a conventional compiler. It is preferable if the compiler produces relocatable code since the memory regions of a process can then be placed anywhere in the single memory space of an M68000 station. Process Manager and the development tools allow for processes of the same type which are mapped to the same station to share code regions, thereby saving memory at the station. Code must be re-entrant to take advantage of code sharing. Although it is not part of the NetKernel paradigm, processes may also share data regions which allows them to interact through shared data. Processes which share data are under the constraint that they must be mapped to a set of stations having common physical memory.

User processes are presently being developed in C with the same compiler that is used for the kernel. However, the use of other languages and compilers is feasible.To make use of a different compiler, a library for performing the NetKernel system calls needs to be provided. These are simply stub functions which set up system call parameters on the stack and trap to the kernel.

3.8.5 NeXT Workstation Prototype

As part of this work, a prototype of the kernel was developed early on. The prototype was developed and runs on a NeXT computer. The NeXT provided a useful environment for testing high-level design ideas, such as the selection and organization of kernel processes. The development effort for the prototype was also reduced by making use of the NeXT's native operating system (Mach with Unix emulation). For instance, it was not necessary to develop context switching code. As with any modern workstation, the operating system (OS) already supports multiprocessing.

From a high-level perspective, the prototype design is quite similar to the M68000 design. The system calls and kernel processes are largely the same. Some incompatibilities are present reflecting minor improvements made to the interface after the prototype was finished. User processes written in C can be compiled without change for either station type if they do not make use of station-specific features. Some of the NeXT services vary from their M68000 counterparts due to the fact that the prototype kernel runs on top of a workstation OS. These differences will be discussed further when the Host Interface Program (HIP) is presented in Chapter 4. HIP needs to recognize the differences since it is a single tool for managing processes in the physical network, which may be heterogeneous.

The prototype design should not be discounted due to the inefficiency of implementing a kernel on top of an OS. Some major benefits compensate for this inefficiency. Far less effort is required compared to an implementation on the bare workstation architecture. The kernel may also be portable to future architectures by the same or other vendors which support the same OS.

Chapter 4

Development Tools: Design and Implementation

4.1 Overview

The development tools provide support for managing the execution of software networks on physical networks. It should be noted that these tools manage *user* software networks. The kernel processes described earlier are present upon start-up at each station. Following start-up, user software networks may be loaded and executed at stations. The development tools support these activities.

NetKernel itself only provides management services at the station level, typically operating on individual processes. These services are replicated at each station in the physical network. The development tools shield the user from having to directly access these services by providing a higher level interface at workstations. One benefit is that control operations at a higher level of abstraction are possible—the user is no longer concerned with the organization and interface of kernel processes. Another benefit is that the low-level interface of the station services is translated to a more attractive format. For example, HIP allows processes to be referred to by user-defined names rather than the numeric process ids (pids) used by Process Manager.

Presently, three tools supporting NetKernel development are available: HIP, Script Generate (SG), and Event Log Report Generator (ELRG). HIP and SG are concerned with the management of software networks. HIP is an on-line tool which supports loading and control of software networks. Commands may be issued to HIP interactively or read from a script file. SG is an off-line tool which generates scripts to load and start software networks. ELRG is an unrelated miscellaneous tool for working with event logs. It produces a text report from the memory image of a NetKernel event log. Description files used for input and output by HIP and SG will be presented in the first section. The three following sections each describe one of the development tools.

4.2 Description Files

Chapter 2 defined software networks, physical networks, and mappings. Three types of description files are used to provide textual descriptions of these entities for manipulation by the development tools: Software Network Description (SND), Physical Network Description (PND), and Mapping Description (MD). SNDs and PNDs are independent descriptions of software networks and physical networks, respectively. An MD describes the mapping of a software network to a physical network. SNDs contain references to an additional type of description file: the Process Type Description (PTD). Each PTD describes the attributes of a particular type of process.

The purpose and content of each description file type are presented in the following three subsections. Information on syntax will not be presented here, but some examples of description files are included in Appendix D. To summarize, each file contains declarations of elements, parameters, and structural information. The SND and PND, for instance, both contain graph descriptions in that they declare nodes and connectivity information. As well, various parameters associated with the nodes and arcs are declared. An inheritance scheme is used whereby each description file may declare a *super-file*. If a piece of information cannot be found in a file, then its super-file is searched. Borrowing again from object-oriented terminology, we use the term *sub-file* to describe the relationship of a file to its super-file.

4.2.1 Software Network Description (SND)

An SND describes a software network. The processes and connections of the software network are declared. The type of each process is declared by specifying a PTD for each process. A PTD is a file independent from the SND. It specifies various attributes of a process such as number of input queues and outlets, priority, and memory requirements. The PTD also declares names for the input queues and outlets of a process. These names are used in the SND for the declaration of connections between processes.

The SND and associated PTDs contain all the information about a software network that is required to load the network, such as object code file names, memory requirements and connections between processes. But no information about the allocation of software components to hardware is specified, this being contained in MDs. As described in Section 2.3.3, a software network may contain kernel processes. So for each process, the SND indicates whether the process should be loaded. Processes which need to be loaded are declared as *new processes*; processes which do not need to be loaded are declared as *service processes*. The term service process reflects the typical role of such processes which is to provide services to software networks which are loaded later. Since kernel processes always exist in advance, they are always declared as service processes.

A PTD defines a process type. These types provide building blocks from which software networks are built. Collections of PTDs may be established to allow the reuse of process types. A software network may contain multiple processes of a particular type. Processes declared to have the same PTD will have identical attributes. Parametrization of processes is not currently supported. A limited way to imitate parameterization would be to send initialization messages containing parameters to processes. Sub-files of a PTD can also be used to describe process types which are variations of a common type.

4.2.2 Physical Network Description (PND)

A PND describes a physical network. Stations, links, I/O devices, and their connectivity are declared. These have already been covered in Section 2.5. Service processes available in the physical network are also declared in the PND. (The MD relates service processes which are expected to exist for a software network to service processes which actually exist in the physical network. The former are declared in the SND, whereas the latter are declared in the PND.)

Two types of service processes may be declared in the PND: *common services* and *unique services*. Common services are replicated at each station of a particular type in the network. A user process making use of a common service will be connected to the local process providing the service. For example, consider declaring the Timer process as a common service. Processes of an SND connected to Timer will then always be connected to their local Timer process. This is usually desirable for Timer since using a nonlocal version involves more message system overhead.

A unique service is available at only one station in the network. When a specific service process is to be accessed by user processes throughout the physical network, it should be declared as a unique service. For example, a device driver for a terminal should be declared as a unique service. Any processes connected to that service will then be connected to the device driver at the station of the terminal. This allows the processes to perform I/O with the terminal, regardless of their locations in the physical network.

4.2.3 Mapping Description (MD)

An MD refers to a particular SND and a particular PND. As described in Section 2.6, a mapping allocates elements of a software network to elements of a physical network. Both the SND and PND assign names to the various elements they contain. The MD declares the mapping using the names from the SND and PND.

There are three types of allocations declared in the MD:

O1 the allocation of user processes to stations,

O2 the allocation of service processes to common services and unique services,

O3 the allocation of nonlocal connections to routes through the physical network.

Items O1 and O2 correspond to item E1 of the primary mapping (Section 2.6.1). Item O3 is part of the secondary mapping for our implementation. The primary mapping was also defined to include lengths of input queues (E2). This and other (secondary) mapping information concerning a process can be placed into its Process Type Description (PTD).

This information, however, will be used for all processes of that type. In order to declare mapping parameters unique for an instance of a process (eg. different queue lengths for different processes of the same type), a sub-file of the PTD can be used. The sub-file is then specified instead of the original PTD for the process declaration in the SND. Such a sub-file may be thought of as containing instance specific information for a process. It is more accurate though, to consider it as declaring a new (related) process type since the sub-file can still be referred to repeatedly in an SND to specify several identical processes. Hence such a sub-file is still considered to be a PTD.

4.3 Host Interface Program (HIP)

HIP is the most important of the development tools. It executes on one or more workstations in the physical network. From each workstation, it provides an independent point of control for managing processes throughout the entire physical network. Commands to HIP may be issued interactively from a command line, as well as read from script files.

The following subsection motivates HIP, describing why it is preferable over direct interaction with kernel processes using Operator Interface. Next, HIP's internal model of the physical and software network is described. The following section presents an overview of commands supported by HIP. Finally, the addition of new command modules for supporting activities other than process management is considered.

4.3.1 Motivation

HIP performs management operations by messaging with kernel processes throughout the physical network. The same operations could be performed by an operator sending messages directly to kernel processes using Operator Interface (Section 3.6.3.5). Advantages of HIP over the direct approach are that it provides a higher level interface, supports the use of description files, and supports a set of commands which are independent of station type.

The HIP interface is higher level than the direct approach in that fewer commands are required to accomplish tasks and the commands having a more readable format, making human errors less likely and easier to detect. The Operator Interface requires that the user explicitly select the destination process for a command, including the route to the process if it is nonlocal. HIP is easier to use since it implicitly determines and selects the kernel process(es) to which messages should be sent. A single HIP command may result in several messages being sent to one or more kernel processes. HIP commands have a mnemonic format while the kernel processes have a hexadecimal format for commands. HIP identifies entities such as processes, stations, and links with user-defined names, rather than numeric codes as used by the kernel processes. The use of mnemonics and names rather than hexadecimal codes is friendlier and also makes some errors easier to detect—for instance, a mistyped mnemonic or name is less likely to have a valid interpretation than a mistyped hexadecimal code. HIP also adds new functionality on top of the kernel processes. One such area is the allocation of memory regions, on which the following subsection elaborates.

By accessing description files, HIP reduces the amount of information that needs to be specified in commands. For example, when a new process is allocated, HIP reads the number of input queues, outlets, priority, and memory requirements of the process from its PTD. A final benefit of HIP is that it can support a set of commands which is independent of station type. This is accomplished by building support for different station types into HIP. Consider a 'Start Process' command to HIP. For an M68000 target station, HIP would send the initial values for the program counter and stack pointer to Process Manager at that station. For a NeXT workstation, HIP would send the name of an executable file. In this fashion, HIP provides transparency of station type to the user.

4.3.2 Internal Network Model

HIP maintains an internal model of the physical network and processes executing on the physical network. Knowledge of the physical network enables HIP to send messages to processes at the various stations of the network. By maintaining an internal list of named user processes, HIP allows the user to refer to processes transparent of their location in the physical network.

HIP's representation of the physical network is based on a PND selected by the user. HIP keeps an internal list of processes executing in the physical network, but it does not explicitly associate processes with particular software networks. This may be added in the future, but for the moment a separate HIP session can be used for each software network to achieve the same effect. A number of parameters are recorded internally for each process. These include a user-defined name, the name of the station on which the process is executing, and the name of the PTD for the process. HIP does not presently make use of SNDs. HIP's knowledge of executing processes comes from two mechanisms. Information concerning kernel processes on different types of stations is programmed into HIP. Information concerning user processes is accumulated and updated by HIP as it receives commands for process management such as allocate, start, kill, and so on.

A HIP session does not initially have any information about user processes executing in the physical network. As a user allocates processes during a session, these processes are added to HIP's internal list. Hence, a HIP session generally has access only to user processes created during that session. In this manner, a number of independent HIP sessions can simultaneously manage processes in the same physical network. Each session is aware of only the processes that it created.

HIP can maintain internal representations of various aspects of the network which are useful for process management. An internal model is maintained of the allocation of memory regions on M68000 target stations (memory regions were discussed in Section 3.6.3.1). When allocating a process at a station, HIP reserves code, data, and stack regions for it. HIP consults its internal model to select appropriate unused regions. Requests to reserve regions are sent to Process Manager. A request to Process Manager may fail because HIP's internal model is stale (does not accurately reflect the station's current memory allocation). This could occur as a result of the activities of other HIP sessions, for instance. In this case, HIP regenerates its internal model for the station by querying the station's Process Manager. HIP then selects different regions for the process and tries again.

4.3.3 Overview of Process Management Commands

Process Management commands are broadly divided into two categories: manipulative and status commands. Manipulative commands change the state of the network. Examples include allocating, starting, connecting, and suspending a process. Status commands report on the state of the network without modifying it. The format for these commands has been included in Section B.3. Of the manipulative commands, the sequence for allocating and starting a software network is particularly important. It is presented in the following subsection. The following two subsections discuss remaining manipulative and status commands, respectively. An example sequence of HIP commands for starting a software network is shown in Section D.3 (the sequence can be automatically generated by Script Generator, which is described later in this chapter).

4.3.3.1 Starting a Software Network

To launch or initiate the execution of a software network, several commands must be issued on behalf of each process in the software network. These commands and the corresponding operations performed by HIP will be discussed. The commands are designed to be independent of the type of station to which the process is mapped. HIP will transparently perform operations appropriate for the type of station. Currently, HIP only supports the management of processes at M68000 target stations. But the following discussion will also consider the issues anticipated for management of processes at workstations running NetKernel on top of a local operating system, as in the case of the NeXT prototype.

The activities required to launch a software network have been divided into four steps.

P1 Allocate processes
P2 Establish connections between processes
P3 Load processes
P4 Start processes

For each step, a corresponding HIP command must be issued for each process in the software network. The commands are Allocate, Connect, Load and Start, respectively. As an example, the Allocate command has the following format:

```
ALLOCATE <user-defined process name> <PTD file name> <station name>
```

The three parameters of Allocate are recorded internally by HIP so that subsequent commands involving the process need not repeat the PTD or station name. HIP will frequently refer to the PTD to look up information about the process. The station name parameter specifies the station at which the process should be allocated. Station names are declared in the current PND for the session.

In response to an Allocate command for a process at an M68000 station, HIP performs several operations, including the following. First, various parameters of the process such as number of input queues are looked up in its PTD and passed in an Allocate command to Process Manager at the station. The memory requirements of the processes are determined from the PTD. Using its internal model, HIP selects unused regions and then sends a Reserve Regions command to Process Manager. HIP looks up the priority of the process and sends a Set Priority command to Process Manager. For a workstation running NetKernel, HIP would transparently issue different commands. For instance, HIP would not need to reserve memory regions since workstations handle their own memory management for code, data, and stack.

The other commands of the start-up sequence are not as involved as Allocate. For a Load command to an M68000 station, HIP sends S record lines containing the object code of the process to S Record Loader at the station. For a workstation, it might be assumed that workstation has direct access to an executable file for the process so no operations would be performed by HIP. For the Start command to an M68000 station, HIP sets up the initial context for the process (start address, stack pointer, and so on) and sends a Start command to Process Manager. For a workstation, HIP would pass an executable file name and send a Start command to Process Manager at the workstation.

4.3.3.2 Other Manipulative Commands

The remaining manipulative commands of HIP are relatively straightforward. The following commands are supported: Continue, Remove, SetPriority, and Suspend. These commands are directly translated to corresponding Process Manager commands.

4.3.3.3 Status Commands

The HIP status commands return information concerning stations and processes. Lists of process and station names can be displayed. More detailed status information is also available. For instance, a list of the connections of a process or the memory model for a station can be requested. Some of the status commands cause HIP to query kernel processes in the physical network; however, other status commands do not require such queries since they return information which is internal to HIP.

4.3.4 Command Modules

HIP has been designed to serve a more general role than simply process management. The code for the process management commands described above is contained in a single *command module* of HIP. Other command modules supporting commands for different activities may be added to HIP. Support for both general activities and specific applications is anticipated. An example of a general activity other than process management which could be supported is debugging. As an example of a command module supporting

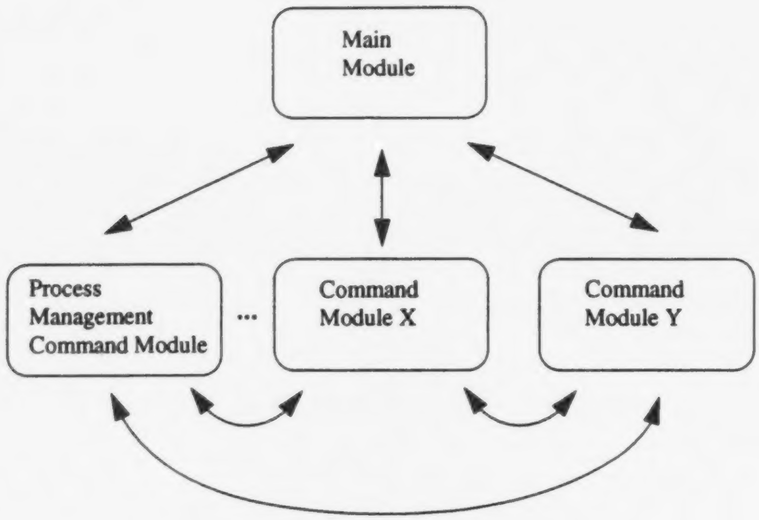


Figure 27 High-level design of HIP

a specific application, we have developed a command module for a simple game. The game, Repulse, is described in Section C.3.

Figure 27 shows the high-level design of HIP in terms of interacting modules. Commands to HIP are passed from the Main Module to the appropriate command module. The command modules make use of some general functions provided by the main module such as functions for sending messages to kernel processes in the physical network. The figure also shows that the command modules interact with each other. Each Command Module should define a programmer's interface for operations which could be potentially useful to other modules. For example, a command module to manage an application with dynamic processes could make use of the Process Management Command Module in order to manage its processes.

4.4 Script Generator (SG)

It was pointed out in Section 4.3.3.1 that initiating the execution of a software network requires several commands for each process in the network. For networks of even just a few processes, it becomes tedious to manually specify these commands. SG solves this problem by automatically generating the required commands. As input, SG requires a SND, PND, and MD. They describe the software network to be executed, the physical network being used, and the mapping of the software network to the physical network, respectively. As output, SG produces a script of commands which can be read by HIP.

Generating the SND, PND, and MD by hand is usually less tedious and less subject to error than generating HIP commands. It also has the advantage that the descriptions may be reused. For instance, given a PND and an SND, one may experiment with different mappings of the software network to the physical network by modifying only the MD. In the future, graphical editors could be used to generate the PND, SND, and MD.

The output script produced by SG contains four sections of commands. Each section corresponds to one of the phases from Section 4.3.3.1: allocate, connect, load, and start. An example of an output script is included in Section D.3.

4.5 Event Log Report Generator (ELRG)

ELRG is a miscellaneous tool unrelated to the ones presented above. It analyzes hexadecimal memory dumps of the event log (Section 3.5.8.1) and produces a formatted text report based on its contents. An example report is included in Appendix E. ELRG can be customized to include the names of user processes and user-defined events in reports.

4.6 Implementation

The tools were programmed in C++ and C. Currently, they run on Sun-3 and NeXT workstations. The Sun-3 workstations do not run NetKernel, so they are included in a physical network as I/O devices (as described in Section 2.5). An object-oriented approach was used for the detailed design and implementation of HIP. The high-level modules shown in Figure 27 were refined in terms of interacting objects.

Chapter 5 Related Work

5.1 Overview

Related work in the field of parallel computer systems will be described and compared to this work. Basic paradigms and architectures have already been considered in Chapter 1. This chapter begins with an overview of environments and paradigms for process oriented concurrent programming. Selected contemporary systems having similar goals to NetKernel are then described. Emphasis is placed on 1) support for the mapping of concurrent programs to hardware, and 2) the runtime environment under which concurrent programs execute. The final section compares NetKernel with the contemporary systems.

5.2 Types of Environments which Support Concurrent Processes

There are two basic levels at which an environment can support the execution of concurrent processes on parallel hardware: runtime software support and direct hardware support. Both approaches provide interprocess communication and synchronization constructs to the programmer. The constructs may be an integral part of a programming language, or they may be added to an existing language (eg. function library).

The runtime software approach entails system software which runs at each processor concurrently with user processes. It makes various services available to users and their programs. Runtime software may be classified as either a kernel or an operating system, depending on the particular services offered. There is no clear-cut boundary, but in general a kernel provides fewer services than an operating system. A kernel might only provide support for multiprocessing with interprocess communication and synchronization. An operating system provides the same functionality and also addresses concerns such as multiple users, protection, file system, I/O, and management of any other resources. Kernels are typically used for single purpose embedded applications, while operating systems are found on general-purpose, multi-user systems such as workstations.

Kernels are particularly suitable for embedded applications, which serve a single purpose over their lifetime. The complexity of an operating system is not justified in this case since many of its features (eg. multi-user support, protection) are not needed. The kernel approach may also prove useful in applying parallel computers to large problems one at a time. Parallel computers may be built from inexpensive processors so simple that they cannot even run full-fledged operating systems. Instead they run kernels. The collection of inexpensive computers may be controlled by more sophisticated computers which do run operating systems.

When each processor in a parallel system runs its own operating system, then the processors generally have more facilities than in the case of a kernel. For instance, each processor will usually have direct access to a local or network file system and hardware support for memory management. Usually, multiple users are allowed to utilize the processors in the network based on their demands and privileges. A network of workstations may be used in this manner. The term *distributed system* is sometimes preferred over parallel computer in this case, to indicate the autonomy of computers in the network and the support for multiple users. In this work, however, it is not important for us to distinguish between the two terms.

The commercial Transputer [Graham90] is unique in that it provides enough hardware support for the execution of concurrent processes on parallel hardware that no runtime software support is required, although additional functionality may be added through software. Most parallel computers, however, are built from processors with less support for parallelism. These processors typically provide hardware support for concurrent processes to execute on a single processor, but provide less support for multiple processors to execute together in parallel. Hence, runtime software is required.

5.3 Influential Concurrent Programming Paradigms

A number of programming paradigms have been proposed for concurrent processes. These will be reviewed since they have influenced the development of kernels and operating systems.

The distinguishing feature of a concurrent programming paradigm in comparison with a sequential paradigm is the provision of facilities for the coordination of processes. Facilities for communication and synchronization are needed. Synchronization includes both mutual exclusion and cross-stimulation of processes.

One of the earliest and most influential constructs proposed for concurrent programming is Dijkstra's *semaphore* [Dijkstra68]. Another early construct for concurrent programming is the *critical section*, for which an explicit programming notation was introduced in [BrinchHan73]. Semaphores and critical sections are low-level constructs such that programming with them is prone to errors. Slightly different versions of a higher level construct, the *monitor*, were proposed in [Hoare74] and [BrinchHan75]. The concept of processes communicating through message passing is formalized in [Hoare78] for which an algebraic analysis technique was presented in [Hoare85]. Hoare's Communicating Sequential Processes (CSP) are synchronized in that both the sender and receiver of a message must be ready for the exchange before either can continue. Such an exchange is referred to as a *rendezvous*. In asynchronous message passing schemes, the sender of a message may continue even if the receiver is not yet ready to accept the message. PLITS (Programming Language In The Sky) [Feldman79] was one of the first such schemes, providing unbounded input queues for processes. Tanenbaum criticizes user level message passing schemes having separate send and receive primitives as "the distributed system's answer to the goto statement: parallel spaghetti programming" [Tanenbaum92, p. 604]. A solution to this criticism is the remote procedure call (RPC) introduced by [BrinchHan73] and further considered in [Birrel84]. Concurrency is also now being considered in the context of object-oriented paradigms, as in [Booch91, pp. 65-68, p. 171].

5.4 Contemporary Systems

Parallel systems which share goals with NetKernel will be considered. For each system, we summarize the hardware architecture and then consider the support for concurrent programs in somewhat greater detail.

In particular, we are interested in systems which address the following concerns:

Q1 hardware which can be arranged into different physical configurations,

Q2 support for the mapping of processes to processors,

Q3 interprocess communication by message passing.

Systems from both research institutions and industry are considered. The common denominator of all the systems considered is that they support the execution of communicating processes on parallel hardware, where software is explicitly formulated as a collection of processes. There are other approaches to parallel software development, such as compilers which automatically parallelize sequential code and programming languages with constructs that lend to parallel execution (eg. pardo). Limiting ourselves to approaches which explicitly identify concurrent processes, we do not consider the other techniques.

5.4.1 Amoeba

Amoeba [Tanenbaum90], [Tanenbaum92, pp. 588-636] provides a transparent distributed operating system. It is also a platform for parallel programming. The hardware model for Amoeba consists of a pool of inexpensive processors (although more complicated processors such as workstations may be used) accessed by users at inexpensive graphics terminals. The justification for this model is that it avoids dedicating expensive and powerful workstations to individual users. It thereby makes all of the computing power available to all of the users.

The Amoeba operating system consists of a microkernel and a collection of server processes. The main functions of the microkernel are process and thread management, memory management, communication, and I/O. Servers also control access to processors, files, and other devices. Threads within a process may synchronize using signals, mutexes, and semaphores. Processes, which may execute on different machines, communicate using RPCs and group communication. Group communication primitives support one-to-many message passing and guarantee that messages are received in the same order by different processes. This particularly simplifies the programming of consensus algorithms and is efficient when the underlying hardware directly supports broadcast between processors. Processes may also communicate through distributed shared memory. Each processor maintains its own copies of memory regions it is sharing. Writes to these regions are handled by broadcasting to all affected processors. Processes are created and managed by RPCs to servers.

5.4.2 CAPER

CAPER [Sugla89] is a programming environment for parallel applications. It is suitable for message passing multiprocessors, with an implementation described for the HPC/VORX [Gaglianell89]. This machine uses a general-purpose architecture where processing power is added in the form of multiprocessors and workstations, and interconnection bandwidth is added in the form self-routing cluster switches arranged in any topology (with an incremental hypercube topology actually in use). CAPER Applications are described by a graph whose vertices represent parallel algorithms and *dataforms* (data transformation routines). These are joined by lines which indicate the flow of data. Hierarchical expansion of the graph is supported. Dataforms restructure and distribute data for parallel algorithms. The number of processors to be used for executing algorithms and dataforms is specified on the graph. A program is described by selecting these elements from libraries and connecting them, with the possibility of coding and adding new elements to the library. The runtime system dynamically establishes channels between elements as they are needed. Data is then transmitted over channels using the underlying message system. A planned extension to CAPER would allow processes to share virtual memory.

5.4.3 CONIC

CONIC [Kramer85,Magee89,Kramer90] supports the development and execution of distributed software in a mixed host and target environment. It is based on a model of asynchronous processes communicating by message passing. CONIC provides two languages: a module programming language and a configuration language. Modules are parameterized sequential programs (i.e. processes) which communicate by sending messages through *exitports* and receiving messages through *entryports*. Message passing is asynchronous. Primitives include send, reply, and receive. The binding between ports is specified using the configuration language. Modules are structured into a hierarchy of *group modules*. The top-level group modules of a distributed application, *logical nodes*, are the basic software units which are allocated to processors. A logical node may be executed as UNIX process on a host machine, or directly under the CONIC executive on a target machine. The configuration language can launch new distributed applications or modify applications which are already executing. CONIC also has facilities to support data conversions for messages sent between machines having different data representations. When required, such conversions are performed automatically at the destination machine.

5.4.4 EMPS

EMPS [Dijk91] provides a set of hardware modules which can be arranged into various physical configurations to build a parallel computer system. A hierarchical scheme is used for the interconnection of processors. The two lowest levels support communication through shared memory over cluster and system buses. LANs are used for the interconnections of system buses. Kernel software handles process management and message passing between processes. Message passing and RPCs are provided transparently between processes at different processors. Message passing uses the rendezvous. Processes executing on processors sharing physical memory may also interact through that memory, including the use semaphores for synchronization and shared data.

5.4.5 Enterprise

Enterprise [Wong92] supports distributed programming for a network of workstations. An ordinary C program is parallelized by graphically associating its function definitions with different icons. Its source code remains unchanged. The types of icons and their interconnection specify the parallelism of a program, such as pipeline and fixed or dynamic pools of worker processes. The Enterprise runtime system creates processes to execute the functions associated with icons. Enterprise inserts extra code for RPCs to pass parameters and return values between processes, thus relieving the user from having to explicitly specify interprocess communication. The runtime system varies the number and mapping of processes based on network conditions and program behavior. New worker processes, for instance, can automatically be started when work piles up for a pool of workers.

5.4.6 Transputer Systems

The Transputer family of processors and an associated software environment are commercially available from INMOS. As well as providing a complete environment for the development of parallel software, these products have been used as a starting point for higher level environments and experimental systems. The first generation of Transputers, some derived works, and the recently announced second generation of Transputers will be considered.

5.4.6.1 First Generation

Transputers [Graham90],[Carlini91] may be arranged into arbitrary networks using high speed serial links, subject to the limitation that each processor has a maximum of four serial ports. Crossbar switches are also available. These are typically used to allow off-line reconfiguration without changing physical connections. It is also possible to dynamically route messages at runtime through a crossbar by setting up an appropriate control system. Transputers provide complete hardware support for the scheduling of processes and message passing between processes. Transputers have been specially designed to execute Occam (more recently Occam2), a concurrent programming language inspired by Hoare's CSP. Channels are declared between processes. Each channel has exactly one sender and one receiver which synchronize (rendezvous) every time a message is exchanged. A major limitation is that only processes at the same Transputer or two Transputers which are directly connected by a serial link may exchange messages. Furthermore, only two channels may be mapped to each serial link, one for each of the link's two directions. Information on the mapping of processes is contained within an Occam program. It should be emphasized the preceding functionality is provided directly by hardware executable instructions. The limitations mentioned have been removed by higher level environments provided by runtime software such as TOPS (to be described shortly).

5.4.6.2 Parallel C

Parallel C [Graham90, pp. 76-84] from 3L Ltd. provides a C programming environment for Transputers. Programs are structured as a collection of tasks. Connections between input and output channels of tasks are described in a configuration file. The configuration file also describes the physical topology of the Transputer network and the mapping of processes to Transputers. A runtime library provides send and receive functions for message passing between tasks. As with message passing under Occam on the Transputer, rendezvous is used. The code of a task contains no fixed references to other processes. Message destinations are specified using local variables which the runtime system initializes. The runtime library also includes functions for thread creation and synchronization. A task may spawn threads dynamically while it is executing, but all threads of a task execute on the same Transputer. Variables may be shared by threads in the same task.

5.4.6.3 TOPS

TOPS [Hubertus90] is a distributed kernel for Transputer systems. It provides a new process model and extends the message passing system. Each process is the sole owner of a collection of ports (message queues) through which it may asynchronously receive messages. Other processes may send messages to these ports. The TOPS kernel provides a higher level message passing model than Occam. RPCs and broadcasting are supported. Additionally improvements over Occam are that processes on non-adjacent stations may communicate and that serial links are no longer limited to only two channels. This allows all mappings of a process network to a transputer network to be realized.

5.4.6.4 T-Rack

T-Rack [Boianov91] is a reconfigurable Transputer network developed at Manchester University. A shared memory subsystem was designed to support higher bandwidth exchange of messages between Transputers. The subsystem is compatible with Occam message passing and therefore supports that programming environment. Transputers are arranged in clusters of four on a single board. Transputers on the same board communicate by passing messages through shared memory, while Transputers on different boards communicate via serial links.

5.4.6.5 Second Generation

The second generation of the Transputer family [Carlini91, pp. 182-206] removes the limitations mentioned earlier (Section 5.4.6.1) through improvements to the hardware, as well as providing numerous other improvements in terms of performance and convenience. Most importantly, it allows all mappings of a process network to a transputer network to be realized (with the addition of an interconnection network). A new processor, the T9000, allows many virtual channels to be multiplexed on a single link. This allows any number of processes, which still must be at adjacent stations, to communicate over a single link. However, interconnection networks may be built between T9000s using the new C104 routing chip. Processes at stations which are not directly linked may communicate using messages which take one or more hops through the interconnection network. All of this support is provided directly in hardware, with the possibility of adding runtime software for further extensions.

5.5 Comparison and Discussion

The preceding systems were selected as ones having major goals in common with NetKernel. Comparisons will now be drawn between these systems with respect to their goals and the approaches taken to achieve them. Architecture, interprocess communication, programming-in-the-large, and mapping will be considered.

The NetKernel hardware model particularly resembles that of Amoeba, Conic, Transputer, and EMPS. In all of these, software may be executed on different arrangements of target processors, which are controlled from one or more hosts. As a distributed operating system, this is mainly a matter of generality for Amoeba. For the others and NetKernel, it is a primary concern that the physical network can be arranged into a configuration selected on the basis of enhancing the performance of a particular concurrent program. This is especially appropriate for embedded systems where a physical network is dedicated to the execution of a single program over its lifetime. In contrast, the processors of an Amoeba system will be arranged to provide good performance in general for a wide range of different programs and users. Enterprise is a tool for making use of idle workstations in a network, so it does not consider target systems. The emphasis for Amoeba, Enterprise, and CAPER is on providing a general-purpose environment for a fixed arrangement of processors, although the actual arrangement may vary from site to site. Any virtual topology can be realized at a site, but the hardware is almost never rearranged to provide a more efficient physical topology for user programs.

Amoeba, T-Rack and EMPS describe hardware in which processors may share memory. Amoeba and EMPS allow processes to communicate directly through shared memory. In EMPS, processes which communicate through shared memory must be mapped to stations which actually share physical memory. Amoeba emulates shared memory between processors when necessary, so no such mapping constraints are introduced. T-Rack does not include shared data in its programming paradigm, but uses the common memory to achieve faster message passing. This is the primary benefit envisioned for NetKernel once shared memory has been implemented. The presence of links of varying performance (i.e. serial links and shared memory) provides a more interesting experimental environment for mapping. Processes which exchange messages at relatively high rates can be clustered onto processors sharing memory for better performance. It is anticipated that NetKernel processes will also be allowed to communicate directly through shared memory in special cases. The decision to do so, however, should not be made lightly since it introduces mapping constraints on processes.

All of the systems including NetKernel make use of message passing to exchange data between processes. Amoeba and Enterprise do not allow user processes to directly send or receive messages, but instead provide RPCs which are built on top of an underlying message system. As Tanenbaum suggests, this results in more comprehensible concurrent programs [Tanenbaum92, p. 604]. The remaining systems make send and receive primitives directly available to the programmer. Except for the Occam and Parallel C Transputer environments, a reply primitive is also available. This allows the realization of client-server relationships and RPCs. Other runtime systems for the Transputer, such as TOPS make reply available on that platform. The send primitive in Conic, TOPS and NetKernel is asynchronous, while it is synchronous in Occam and Parallel C. Amoeba also supports group communication. Of the other systems considered, only TOPS has any support for one-to-many message communication, providing a broadcast primitive. However, one-to-many communication between processes is also possible on EMPS (and Amoeba) through shared data.

CAPER, CONIC, NetKernel, and Parallel C support programming-in-the-large [DeRemer76]. This allows concurrent programs to be built by specifying connections between software components. No changes need to be made to the source code of components in order to incorporate them into different programs, unless the functionality needs to be changed. A new component may be developed when no satisfactory component already exists. An essential feature of these systems is that the code of a process contains no direct references to the processes with which it communicates. Message passing operations are specified in terms of local entities of processes, such as input queues and outlets (NetKernel), entryports and exitports (CONIC), or CHAN variables (Parallel C). A graph description contained in a configuration file is used to describe how processes are connected together to form a concurrent program. CAPER and CONIC allow for hierarchical descriptions, which NetKernel does not presently support. The runtime system is responsible for establishing the specified connections between processes so that they may communicate. The other systems contain explicit references to processes within the code of processes, so they do not directly support the construction of concurrent programs by interconnecting library components without the modification of any source code. For RPCs, particularly as in Amoeba and Enterprise, users may make use of library components simply by including the appropriate function calls in their code. Other approaches such as named channels (Occam) and process identifiers (EMPS and TOPS) require greater effort to maintain the appropriate linkages between processes in their code, although runtime schemes to initialize variables with identifiers as in Parallel C could be added. Some advantages of allowing user code to directly manipulate process identifiers are that it 1) provides a way to identify processes dynamically spawned by user processes at runtime, and 2) allows references to processes to be easily exchanged between processes. Because CAPER, CONIC, NetKernel, and Occam are concerned with mainly static networks of processes, these issues are not major concerns.

Occam, CONIC, Parallel C, and NetKernel provide several different, but equally powerful, mechanisms for a process to wait non-deterministically until a message arrives on any channel of a set of input channels. CONIC uses a select statement in which alternative channels and corresponding actions are declared. The Occam ALT statement is similar. Parallel C requires that a process associate a different thread with each channel. The NetKernel Wait() call specifies alternatives as terms of a disjunction. The use of boolean expressions relating to input queues for a blocking wait primitive is unique to NetKernel. It allows synchronization of several channels to be expressed particularly compactly. This feature has been included to specially support the future exploration of programs compiled from a design notation based on Petri nets (Section 6.2.1).

Amoeba, CAPER and Enterprise automatically select processors for processes at runtime, with optional provisions for users to explicitly specify processors when desired. The automatic selection of least loaded stations is highly appropriate since these are general-purpose systems where multiple users are likely to run programs at the same time. Amoeba and Enterprise monitor loading throughout the network and select the least loaded stations. Conic, EMPS, NetKernel and the Transputer require that stations be selected for each process prior to runtime (i.e. mapping). This approach is appropriate when a system is being used to execute a single program. In this case, no prior load is present to influence the selection of stations. It is then more appropriate to base the mapping on structural and performance matching between the concurrent program and target hardware.

Of the systems considered, CONIC, CAPER, NetKernel and Parallel C provide environments for programming-in-the-large, allowing programs to be constructed by interconnecting software components. Program code is associated with components (i.e. processes) and need not be changed when components are reused in different programs. Of these, CAPER differs in that it does not promote a reconfigurable physical network nor does it encourage the user to explicitly select the mapping for concurrent programs. NetKernel bears the most similarity to CONIC and Parallel C. Of these, Parallel C is distinguished by synchronous message passing and lack of a reply primitive, needed for general client-server relationships. Parallel C and NetKernel do not support hierarchical description of the process graph as CONIC does. But they do provide finer control over the mapping of processes, allowing the target station for each process to be individually identified. In CONIC, the top level components of the hierarchical description are mapped to stations. All sub-components of a top level component are therefore mapped to the same station. NetKernel emphasizes flexibility in mapping by separating the descriptions of the physical network, software network, and mapping. CONIC and Parallel C mix aspects of these descriptions into a single configuration file (although it would be relatively trivial to generate the single file from separate descriptions).

Chapter 6 Conclusions

6.1 Conclusions

Chapter 1 motivated and established the basic approach for NetKernel. An asynchronous, message passing network of processes executing on parallel hardware was selected. The NetKernel paradigm was presented in Chapter 2. In particular, the concepts of software network, physical network, and mapping were defined. Chapter 3 described the design and implementation of the NetKernel paradigm on M68000 target stations available in department. The NetKernel software runs on each station in the physical network. Its design consists of two parts: a microkernel and a collection of kernel processes. Together, these provide facilities for local process management and transparent distributed interprocess communication. Chapter 4 presented the design and implementation of workstation based development tools which are used in conjunction with NetKernel. HIP and Script Generator support loading, execution, and control of a software network on a physical network. Chapter 5 described some systems with similar goals to NetKernel and compared these approaches.

The paradigm, design, and implementation of NetKernel have been presented. This constitutes the initial effort in the development of an experimental platform for concurrent software. Some simple test programs presented in Appendix C demonstrate that the operational requirements (Section 2.2) for NetKernel have been achieved. These requirements were to support interprocess communication and synchronization, and to provide flexibility in the mapping of concurrent software to parallel hardware. The ability to achieve different levels of performance through different mappings has been demonstrated (Section C.2). Although our platform is not high performance in any sense, it does provide speedup for certain granularities of concurrent programs. We are confident that it can be used to explore some general issues in concurrent systems that will be applicable to more state of the art parallel computers. In particular, a primary motivation for NetKernel is to provide a software environment for the execution of programs derived from a design notation based on Petri Nets (discussed further in Section 6.2.1). Having achieved our operational requirements, NetKernel now provides such an environment.

6.2 Future Work

There are many possibilities for extending and improving NetKernel. But NetKernel is now ready to serve its intended purpose as a software environment for the execution of higher level net-oriented paradigms. A design notation based on Petri Nets is of particular interest. This should now be pursued. Then, based on experience in that effort, the most pertinent extensions and improvements can be determined and added to NetKernel. Some ideas concerning our approach using Petri Nets will be presented. The related areas of performance prediction and mapping are then considered. Finally, an assortment of possible extensions and improvements to NetKernel are discussed.

6.2.1 Design Notation Based on Petri Nets

We are interested in using NetKernel to explore a design notation based on Petri Nets. The flexibility of mapping provided by NetKernel and the form of the wait() system call were conceived specifically to support this endeavour. The motivation for a design notation based on Petri Nets is discussed below. In order to execute these designs in software using NetKernel, a compiler is required which outputs NetKernel programs from J-net designs. We consider some issues of this compilation.

For a general introduction to Petri Nets, the reader is referred to [Murata89]. Petri Nets are well known as a powerful modeling tool for representing concurrency and non-determinism. The design notation we are considering, J-nets, has been proposed in [Joerg91]. To briefly summarize the notation, processing functions are carried out in places of the net. Places may also be used purely for the storage of tokens. Places are interconnected by arcs and transitions. Data tokens and control tokens flow along arcs. Transitions serve to synchronize the flow of tokens by firing when all incoming tokens are available. Arcs may also provide control operations, such as resetting a place to empty. Design using J-nets requires 1) formulation of a J-net, and 2) description of the processing performed by places in the net. The J-net provides an overall structural description of the design. The processing functions may be described in various ways, such as program code, logic, or circuit designs.

One of the motivations behind J-nets is *scalability of design*. Once a design has been expressed as a J-net, it is mapped to physical resources for implementation. The design may be adjusted to meet different constraints (eg. performance) through a new mapping without any need for redesign. J-net notation is suitable for both hardware and software, so mapping may include the allocation of processing functions to both dedicated VLSI circuitry and to programs executing on processors. Compilers will be used to generate the circuitry and/or software associated with a mapped design. An exciting aspect of this approach is that a design expressed as a J-net may be compiled into a mixed implementation of custom digital circuitry and software. The compilers could automatically generate the interface for exchanging information across the hardware/software boundary. The boundary may be adjusted through different mappings.

Figure 28 illustrates a direct compilation of a Petri Net fragment into a NetKernel program (note that J-net notation has not been used here—ordinary Petri Nets will suffice for these examples). The approach taken is to convert each place in the net to a corresponding NetKernel process, referred to as a *place-process*. Each conflict set (of transitions) of the net

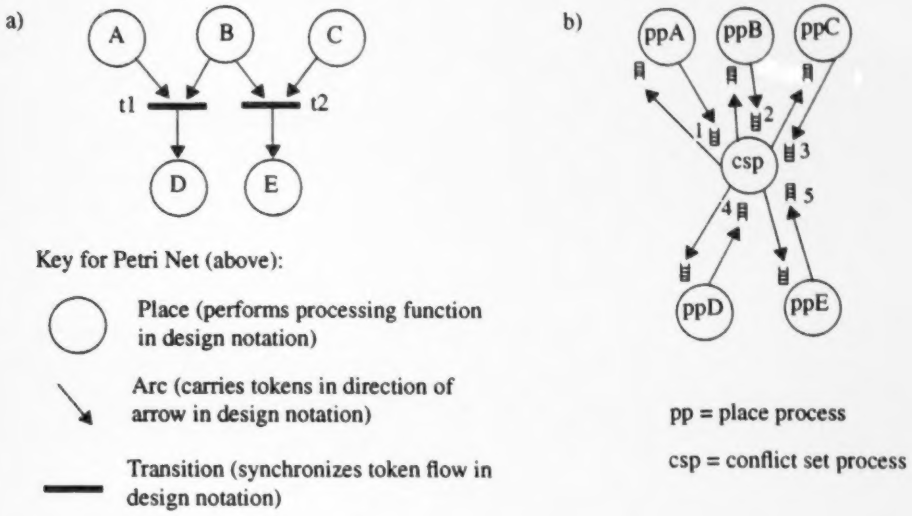

a)
A
B
C
t1
t2
D
E
Key for Petri Net (above):
Place (performs processing function in design notation)
Arc (carries tokens in direction of arrow in design notation)
Transition (synchronizes token flow in design notation)
b)
ppA
ppB
ppC
1
2
3
csp
4
5
ppD
ppE
pp = place process
csp = conflict set process

Figure 28 Compiling a Petri Net design description into a NetKernel software network

a) A simple Petri Net

b) A corresponding NetKernel software network.

is also converted to a corresponding process, referred to as a *conflict-set-process*. A conflict set is a set of transitions whose firing may conflict with each other. The flow of tokens in the net directly corresponds to the flow of messages in the software network. In this example, the two transition may conflict, so they are mapped to a single conflict-set-process, *csp*, in Figure 28b.

Assuming that the processing functions of the Petri Net places are described by code segments, the compiler would insert these segments into code templates which handle the message operations associated with token flow. These templates then provide the code for the place-processes of the software network. The code for transition sets is automatically generated to preserve the semantics of the transitions contained in the design notation. In Figure 28, the software network contains outlets corresponding to the directed arcs of the Petri Net and also in the *reverse* direction of the arcs. This allows for flow control in software networks. Places in J-Nets, for instance, have a capacity of 1 token. A process-place sends a message to its source conflict-set-process to indicate when it is able to accept another token.

Petri Net semantics imply that a transition fires whenever all of the places connected by arcs *towards* the transition (i.e. input places) contain tokens. An additional limitation may be imposed that all output places of the transition must be empty. The NetKernel system call wait() has been specially designed to support these semantics for conflict-set-processes. Using the compilation scheme of Figure 28, the conflict-set-process receives messages corresponding to tokens from all input places. It also receives "ready" messages from empty output places. By using the disjunctive normal form of the wait() call, the process can be activated when any of its associated transitions are ready to fire. For Figure 28, *csp* would use the system call wait($(1\wedge2\wedge4)\vee(2\wedge3\wedge5)$) to wait until either *t1* or *t2* is ready to fire. Satisfaction of the left term indicates that *t1* can fire, whereas the right term indicates that *t2* can fire.

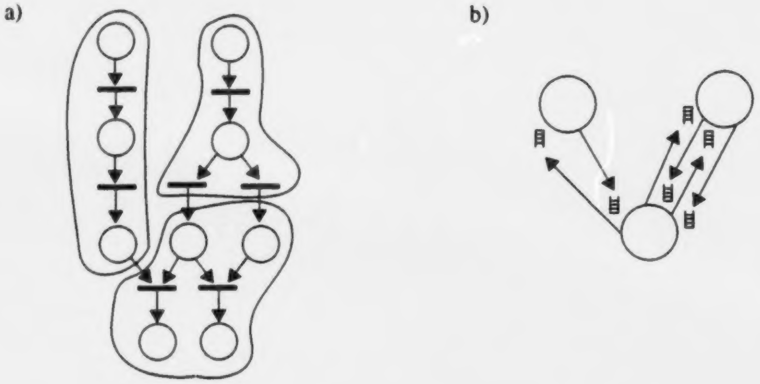


Figure 29 Compiling by collapsing

The preceding example illustrates a very direct compilation from J-Net design to NetKernel program. This scheme results in NetKernel programs with high overhead if many processes are mapped to the same station. The overhead results from the exchange of messages and context switching between processes. A more efficient approach would be to collapse portions of a net into processes, as shown in Figure 29. Criteria for selecting the boundaries need to be determined. One reasonable criterion is that all of the transitions of a conflict set should be collapsed into the same NetKernel process.

[Bütler90] describes a distributed Petri Net simulator implemented using Occam on Transputers. A variant of Predicate/Transition nets [Genrich87] is used which allows data to be associated with tokens and operated upon inside transitions (in contrast to J-nets, where processing is associated with places). A separate process is used for each place and each transition. A distributed scheme arbitrates between conflicting transitions. The Transputer's low overhead for context switching and message passing is conducive to this approach. Bütler has developed configuration tools which select mappings and hardware configurations using a simulated annealing algorithm.

6.2.2 Performance Prediction and Mapping

NetKernel can be used as a tool to verify performance prediction and mapping techniques for concurrent software on parallel hardware. Performance prediction would provide an estimate of the performance of a software network for a particular mapping to hardware. The performance might be described in terms such as throughput, response time, or total execution time. An accurate prediction of performance avoids the need to experimentally measure the performance of mappings, particularly bad mappings which do not come close to meeting objectives.

Mapping techniques select a mapping which satisfies performance objectives on available hardware. Having designed a software network, for instance, we might wish to determine a mapping which provides a certain throughput upon any configuration of available hardware whose total cost does not exceed some figure. Ideally, a mapping technique would either determine a satisfactory mapping and physical configuration, or indicate that none exists. The general mapping or *module allocation* problem is shown to be NP-complete in [Fernandez89] (some special cases which do have polynomial time solutions are also discussed). As a result, heuristic methods for mapping are being proposed as in [Sinclair87].

Performance prediction and mapping are general issues in concurrent software. These investigations are relevant with respect to the Petri Net design notation described previously. The scalability of J-net designs rests on being able to map J-nets to the physical resources available for implementation, whether those resources are custom circuitry, processors executing software, or both.

6.2.3 Improvements and Extensions to NetKernel

Various improvements and extensions which could be made to NetKernel will be considered. The subjects of additional hardware, hierarchical software network descriptions, graphical interfaces, improving serial link performance, error recovery, new message system primitives, flow and congestion control, and process migration are touched upon.

Support for new hardware in the physical network would allow for more interesting mapping problems to be considered. MPAXS is ultimately envisioned as a scalable, reconfigurable parallel computer built from hardware components with different levels of performance. To this end, work is currently underway to develop a shared memory subsystem allowing up to four M68000 units to access a common memory. The memory will be used for higher bandwidth message passing between the units than is possible over serial links. The incorporation of a new link type requires that a new Link Device Driver (LDD) process be written and added to the kernel.

Eventually, the kernel should be ported and/or rewritten for different processor types (both target and host). This would provide a heterogeneous, mixed host/target environment. The Repulse game (Section C.3) is an initial demonstration of a mixed host/target NetKernel system. But the Next prototype has not been fully integrated with the development tools, particularly the Host Interface Program (HIP). Developing NetKernel on the Sun workstation would be quite useful since the Sun could then be incorporated into physical networks as a station rather than just an I/O device. This would make it possible to run multiple HIP sessions from a single Sun, as well as providing a mixed environment. Different processor types often have incompatible data formats. The issue of data conversion needs to be considered. One approach would be to insert conversion processes at appropriate places in the software network.

The description of software networks could be extended to allow the use of hierarchy. It would be convenient if a description of a network of processes could be encapsulated and then included into other designs simply by referring to the encapsulated description. It is unclear whether the hierarchy should be flattened at runtime or maintained by the kernel.

Currently software network descriptions, physical network descriptions, and mapping descriptions are created manually using a text editor. Graphical editors would provide a more convenient way of generating these descriptions.

The packet format used for messages over the serial links could be optimized to eliminate a few bytes, resulting in less overhead. The network layer header could be changed to allow for packets sizes of greater than 255 bytes. Running the serial links at a rate higher that 9600 baud could improve their packet throughput, although the cpu overhead of servicing interrupts would soon become the limiting performance factor.

Error handling and recovery need to be considered further. Allowing messages to be prioritized could be useful here. Currently, there is no way for a nonlocal message to receive preferential treatment as it propagates through the physical network. It is possible for processes to offer preferential treatment to arrived messages by the order in which they service their input queues.

More message passing primitives could be added. Group communication or at least some primitive for one-to-many message communication could be added. A simple although inefficient alternative for one-to-many communication is to develop a library of "broadcaster" processes which receive messages and then send them to many destinations. These could then be included in software networks whenever one-to-many communication is needed. A process was used for this purpose in the Repulse game (Section C.3).

NetKernel provides no direct treatment of flow control between processes or congestion control of network traffic (Section 3.4.2). The current approach is that congestion is avoided through choice of an appropriate mapping. Flow control may be handled similarly or mechanisms may be built into processes as in SYNTH (Section C.2). Or flow control could be added to NetKernel. The obvious approach of flow control between all pairs of communicating processes introduces more overhead than necessary. SYNTH, for instance, is more efficient in that it uses only one outer feedback loop for flow control of the entire software network (Figure 31b).

Process migration could be added, although it would be terribly slow over our current serial links. Higher bandwidth links such as shared memory might make this more feasible. The code for processes could also be placed into shared memory, making no transfer of code necessary to migrate processes between stations sharing memory. The drawback of this approach is that there is much more contention for shared memory when it contains program code. For this reason it is preferable to use shared memory only for the exchange of messages (and possibly for data shared between processes).

Some features lacking from NetKernel which would make it a more attractive environment for programmers have been mentioned above. It should be kept in mind, however, that NetKernel has been designed to execute on simple processors having very little memory. The benefits of new features should be weighed against their cost in terms of memory and cpu utilization. The approach until now has been to minimize the functionality required at target stations, trading it off for increased complexity at the hosts where possible (eg. Host Interface Program). Since NetKernel is intended to execute programs compiled from net descriptions, it is of little value to add functionality for the convenience of a NetKernel programmer. If this emphasis shifts and programs are being developed directly for NetKernel, then such improvements would be more justified.

Bibliography

Ambler92	A.L. Ambler, M.M. Burnett, B.A. Zimmerman, "Operational Versus Definitional: A Perspective on Programming Paradigms", IEEE Computer, Vol. 25, Num. 9 (Sep. 1992), pp. 28-42.
Barrett92	A.T. Barrett & Associates, Inc., *MCX-16 MicroController eXecutivefor the Motorola M68HC16Z1 Evaluation Board Version 1.0*, A.T. Barrett & Associates, Inc., Houston, TX, 1992.
Ben-Ari82	M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall, 1982.
Birrel84	A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, Num. 1 (Feb. 1984), pp. 39-59.
Boianov91	L.K. Boianov, A.E. Knowles, "Higher Speed Transputer Communications Using Shared Memory", Microprocessors and Microsystems, Volume 15, Number 2 (Mar. 1991), pp. 67-72.
Bütler90	B. Bütler, R. Esser, R. Mattmann, "A Distributed Simulator for High Order Petri Nets", Advances in Petri Nets 1990, Springer-Verlag, Heidelberg, 1991, pp. 47-63.
BrinchHan73	P. Brinch Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
BrinchHan75	P. Brinch Hansen, "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol. SE-1 (June 1975), pp. 197-207.
BrinchHan78	P. Brinch. Hansen, "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, Vol. 21, Num. 11 (Nov. 1978), pp. 934-941.
Booch91	G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings Publishing Company, Redwood City, California, 1991.
Carlini91	U. De Carlini, U. Villano, *Transputers and Parallel Architectures*, Ellis Horwood Limited, West Sussex, England, 1991.
Courtois85	P.-J. Courtois, "On Time and Space Decomposition of Complex Structures", Communications of the ACM, Vol. 28, Num. 6 (June 1985), pp. 590-603.
Day83	J. Day, H. Zimmerman, "The OSI Reference Model", Proceedings of the IEEE, vol. 71 (Dec. 1983), pp. 1334-1340.

DeRemer76 F. DeRemer, H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small", IEEE Transactions on Software Engineering, Vol. SE-2, Num. 2 (June 1976), pp. 80-86.

Dijkstra68 E.W. Dijkstra, "Cooperating Sequential Processes", editor F. Genuys, *Programming Languages*, Academic Press, New York, 1968.

Dijk91 G.J.W. Dijk, A.J. van der Wal, "EMPS: The Design of an Architecture for a Distributed Homogenous Multiprocessor System", Microprocessors and Microsystems, Vol. 15, Num. 4 (May 1991), pp. 187-194.

Duncan90 R. Duncan, Control Data Corporation, "A Survey of Parallel Computer Architectures", IEEE Computer, Vol. 23, Num. 2 (February 1990), pp. 5-16.

Feldman79 J.A. Feldman, "High Level Programming for Distributed Computing", Communications of the ACM, Vol. 22, Num. 6 (June 1979), pp. 353-368.

Fernandez89 D. Fernández-Baca, "Allocating Modules to Processors in a Distributed System", IEEE Transactions on Software Engineering, Vol. 15, Num. 11 (Nov. 89), pp. 1427-1435.

Flynn72 M.J. Flynn, "Some Computer Organizations and their Effectiveness", IEEE Transactions on Computers, Vol. C-21, Num. 9 (Sep. 1972), pp. 948-960.

Gaglianell89 R.D. Gaglianello, B.S. Robinson, T.L. Lindstrom, E.E. Sampieri, "HPC/VORX: A Local Area Multicomputer System", IEEE 9th International Conference on Distributed Computing Systems, 1989, pp. 542-549.

Genrich87 H.J. Genrich, "Predicate/Transition Nets", Lecture Notes in Computer Science 254, Petri Nets: Central Models and their Properties, Springer-Verlag, Heidelberg, 1987, pp. 207-247.

Gomaa84 H. Gomaa, "A Software Design Method for Real-Time Systems", Communications of the ACM, Vol. 27, Num. 9 (September 1984), pp. 938-949.

Graham90 I. Graham, T. King, *The Transputer Handbook*, Prentice-Hall International (UK), 1990.

Hoare74 C.A.R. Hoare, "Monitors, An Operating System Structuring Concept", Communications of the ACM, Vol. 17, Num.10 (Oct. 1974), pp. 549-557. Erratum in Vol. 18, Num. 2 (Feb. 1975), p. 95.

Hoare78 C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM, Vol. 21, Num. 8 (Aug. 1978), pp. 666-677.

Hoare85 C.A.R. Hoare, *Communicating Sequential Processes*, Englewood Cliffs, NJ, Prentice-Hall, 1985.

Hubertus90 F. Hubertus, B. Abbott, "TOPS A Distributed Operating System Kernel for Transputer Systems", Proc. of the 22nd Southeastern Symposium on System Theory, IEEE, Cookeville, TN, 1990, pp. 103-107.

Kramer85 J. Kramer, J. Magee, "Dynamic Configuration for Distributed Systems", IEEE Transactions Software Engineering, Vol. SE-11, Num. 4 (Apr. 1985), pp. 424-435.

Kramer90	J. Kramer, J. Magee, A. Finkelstein, "A Constructive Approach to the Design of Distributed Systems", IEEE 10th International Conference on Distributed Computing Systems, 1990, pp. 580-587.
Loepere90	K. Loepere, *Mach 3 Server Writer's Guide*, Open Software Foundation and Carnegie Mellon University, 1990.
Joerg91	W.B. Joerg, "A Petri-Net Based Approach to the Synthesis of Parallel Asynchronous Processing Structures", 4th International Forum on ASICs and Transducers, Leuven, May 1991, pp. 13-28.
Magee89	J. Magee, J. Kramer, M. Sloman, "Constructing Distributed Systems in Conic", IEEE Transactions on Software Engineering, Vol. 15, Num. 6 (June 1989), pp. 663-675.
Motorola89	Motorola Inc., *M68000 User's Manual*, 6th edition, Prentice-Hall, 1989.
Murata89	T. Murata, "Petri Nets: Properties, Analysis, and Applications", Proceedings of the IEEE, Vol 77, Num. 4 (Apr. 1989), pp. 541-580.
Poplawski91	D.A. Poplawski, "Synthetic Models of Distributed-Memory Parallel Programs", Journal of Parallel and Distributed Computing, Vol. 12, Num. 4 (Aug. 1991), pp. 423-426.
Quinn87	M.J. Quinn, "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill, New York, 1987.
Silbersch89	A. Silberschatz, J.L. Peterson, *Operating System Concepts*, Alternate Edition, Addison-Wesley, 1989.
Sinclair87	J.B. Sinclair, "Efficient Computation of Optimal Assignments for Distributed Tasks", Journal of Parallel and Distributed Computing, Vol. 4, Num. 4 (Aug. 1987), pp. 342-362.
Software88	SOFTWARE DEVELOPMENT SYSTEMS, INC., "CrossCode C for the 68000 Microprocessor Family", 1988.
Stankovic88	J.A. Stankovic, K. Ramamritham, "Introduction, 1.4: Emerging Principles, Abstractions, and Primitives", Tutorial Hard Real-Time Systems, IEEE Computer Society Press, 1988.
Sugla89	B. Sugla, J. Edmark, B. Robinson, "An Introduction to the CAPER Application Programming Environment", IEEE 1989 International Conference on Parallel Processing, 1989, part II, pp. 107-111.
Tanenbaum89	A.S. Tanenbaum, *Computer Networks*, 2nd ed., Prentice-Hall, 1989.
Tanenbaum90	A.S. Tanenbaum, "Amoeba A Distributed Operating System for the 1990s", IEEE Computer, Vol. 23, Num. 5 (May 1990), pp. 44-53.
Tanenbaum92	A.S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992.
Whiddett87	D. Whiddett, *Concurrent Programming for Software Engineers*, Chichester, England, Ellis Horwood Limited, 1987.
Wong92	P. Wong, *The Enterprise Executive*, Technical Report TR92-13, Dept. of Computing Science, University of Alberta, Edmonton, Canada, 1992.

Appendix A Glossary

Connection
A connection joins an outlet of a process to an input queue of a different process. Connections are established dynamically at runtime, usually after the processes of a software network have been loaded and before they are started. When a process sends a message through one of its outlets, the message gets deposited into the input queue to which the outlet is connected. Note: As used here, the term connection does not have the same implication as in telecommunications, where it implies that communication resources or bandwidth have been reserved between a sender and receiver.

Handler Process
A handler is a special kind of process on M68000 targets stations which can service exceptions. Hence they are used to implement device drivers. Handlers are subject to certain restrictions compared to *ordinary* processes in order to reduce the overhead associated with exception servicing.

Host
A host is typically a workstation running an operating system with access to a file system. Hosts are used for running development tools and controlling physical networks. Contrast to *target*. Currently Sun-3 and NeXT workstation hosts are supported.

I/O Device
I/O devices are attached to stations in the physical network. NetKernel programs may receive input from an I/O device or send output to it through a device driver process running at the same station. Examples include terminals, printers, and disk drives (of these, only terminals are currently in use). Processors which do not execute NetKernel may also be included in the physical network as I/O devices. This is useful for workstations and special purpose processors, and is currently being used for Sun workstations.

Kernel Process
Kernel processes are part of NetKernel. They are started at the same time as the kernel. They are contained in either ROM or start-up files, depending on the implementation.

Link
A link is hardware which allows direct communication between two or more stations. Links are classified as either point-to-point or multiple access.

Mapping
A mapping is an allocation of the (abstract) elements of a software network to the (concrete) elements of a physical network.

Multiple Access Link
A multiple access link provides direct communication between two or more stations. This link is distinguished from a point-to-point link in that the sender must explicitly select a receiver. Ex. common memory, Local Area Networks (Ethernet, token ring, etc.).

Operator
An operator is a person who controls the execution of NetKernel software networks.

Ordinary Process
An ordinary process executes under the NetKernel paradigm with access to all the primitives defined by the paradigm. This is in contrast to a *handler* process, which is a restricted form of process available on M68000 targets.

Physical Network
A physical network is an arrangement of interconnected hardware. A physical network contains stations, links, and I/O devices.

Point-to-Point Link
A point-to-point link provides direct communication between exactly two stations. These links are distinguished from multiple access links in that there is no need for the sender to explicitly select a receiver on the link. Ex. RS232 serial link.

Process
A process is a sequential program in execution. NetKernel associates a set of input queues and a set of outlets with each process. Processes are classified in two ways: user vs. kernel and ordinary vs. handler (M68000 only).

Software Network
A software network contains processes and connections. Connections join input queues of processes to outlets of processes.

Station
A station is a (single processor) computer system which runs the NetKernel software.

Target
A target is a relatively simple (single processor) computer system. Targets do not usually run an operating system and may be as simple as a processor, memory, and minimal peripherals (ex. a communications interface peripheral). Targets are useful in a physical network for guaranteeing real-time response and providing inexpensive computational power. Contrast to *host*. Currently M68000 targets are supported.

User Process
User process are developed by users, in contrast to kernel processes.

Appendix B Selected Aspects of the NetKernel Interface

This appendix provides some details on our M68000 implementation of NetKernel. Section B.1 presents the message passing system calls available to NetKernel processes. The command interface for one of the kernel processes, Process Manager, is summarized in Section B.2. The Host Interface Program (HIP) commands for process management, summarized in Section B.3, provide a higher level interface for managing processes.

B.1 Function Prototypes for Message Passing System Calls

The library MP0.a provides a basic but complete set of system calls. Prototypes for the message passing calls of this library are shown below. Other calls for dynamic memory allocation, generating console messages, etc. have been omitted for brevity. Functions from this library are prefixed by 'MP0_'.

The library MP1.a builds on these calls to provide higher level calls with a more convenient interface. For instance, MP1.a adds timeout options and composite message operations (eg. wait-and-take, RPC). Calls of this library are prefixed by 'MP1_'. Only three of these calls have been shown for brevity.

```
// Send a message through an outlet. Parameters (below) will be summarized for this call only.
/* Parameters: outlet is the outlet number; data points to the message; length is message length in bytes;
freeFlag indicates whether the message memory is still needed by the caller process; replyQid is the queue
number of the caller to which replies should be sent; confirmQid is presently unused, but may eventually
support confirmation of delivery; messageNumber is used to identify replies to the message being sent. */
int MP0_sendOutlet(int outlet, void *data, int length, int freeFlag, int replyQid, confirmQid, int
*messageNumber);

// Reply to a previously received message. The header taken with the previous message is passed.
int MP0_reply(void *header, void *data, int length, int freeFlag, int replyQid, confirmQid, int
*messageNumber);

// Wait until an expression (which refers to the input queues of the caller) is satisfied.
int MP0_waitExpression(struct waitExpression *w);

// Take a message from an input queue.
```

int **MP0_take**(int queueMask, int policy, void **header, void **data, int *length, int *qid, int *replyToNumber, int *replyFlag);

// Examines input queues of caller specified in examineMask, returning those which are ready in readyMask.
int **MP0_examine**(int examineMask, int *readyMask)

// Handler processes use MP0_enable() and MP0_disable() instead of MP0_wait(). These two calls
// enable and disable calling the handler's service procedure when specified input queues are nonempty.
int **MP0_disable**(int queueMask);
int **MP0_enable**(int queueMask);

Selected Calls from MP1.a

// Wait with optional timeout. Messaging with the Timer process is used within the call.
int **MP1_waitExpression_t**(struct waitExpression *w, int timeout, int temporaryQid);

// Shortcut to wait for and take a message. But the wait condition is limited to the form $M_1 \vee M_2 \vee ... \vee M_n$.
int **MP1_wait_take**(int queueMask, int policy, int timeout, int temporaryQid, void **header, void **data, int *length, int *qid, int *replyToNumber, int *replyFlag);

// Remote procedure call (send a message through an outlet, wait for a reply, and take the reply).
int **MP1_rpc**(int outlet, void *data, int length, int freeFlag, int temporaryQid, int replyQid, int timeout, void **replyHeader, void **replyData, int *replyLength, int *messageNumber);

B.2 Command Interface of the Process Manager

The Process Manager at each station accepts command messages in an ASCII format, consisting mainly of hexadecimal numbers. For every command received, Process Manager returns a reply message. The reply message starts with a two digit error code, followed by any additional information returned by the command. An error code of '00' indicates that the command was successfully performed.

An explanation and example are provided for the first command. The remaining commands of Process Manager are simply listed along with their format. Other kernel processes accept different commands using a similar format. Process Manager and Debugger are the two processes with the largest repertoires of commands. Parameters marked with '*' may be repeated zero or more times.

Allocate: 00 <forcePid> <numberOutlets> <numberInputQueues> <queueLength/queuePolicyCode>*

Explanation: Allocates a new process with the specified number of input queues and outlets. For each input queue, its length and a policy code indicating the action upon overflow is specified. A particular process id (pid) may be requested for the process in <forcepid>. If <forcepid> is zero, then Process Manager selects an unused pid.

Example: 00 00 02 04 0300 0300 0600 0700. Allocates a new process with 2 outlets and 4 input queues. The queues have lengths of 3, 3, 6, and 7 messages, respectively. All queues are declared to use the default policy, '00', which discards messages arriving at a full input queue. The outlets and input queues are both referred to using id numbers starting at zero in other commands below.

Change Priority of Process: 05 <pid> <newPriority>

Connect Outlet to Input Queue: 03 <sourcePid> <sourceOutlet> <destPid> <destQid> <numberHops> <hopVirtualLinkNumber>*

Continue Process: 03 <pid>

Kill Process: 04 <pid>

List Processes: 08

Report Connection of Process Outlet: 0D <pid> <outlet>

Report Memory Regions of Process: 47 <pid>

Report Status and Name of Process: 09 <pid>

Report Outlets and Input Queues of Process: 0A <pid>

Report User-Defined Codes of Process: 0C <pid>

Report User Memory Space: 48

Reserve Memory Regions of Process: 46 <pid>

Set Initial Context of Process: 40 <pid> <initialPC> <initialUSP> <initialA5> <initialA4>

(The Debugger process can be used to define the initial context more completely when needed).

Set Process Name: 0E <pid> <processName>

Set User-Defined Code of Process: 0B <pid> <codeNumber> <codeValue>

Start Process: 01 <pid>

Suspend Process: 02 <pid>

B.3 HIP Commands for Process Management

The Host Interface Program (HIP) builds on the Process Manager commands to provide a higher level interface for the management of processes. HIP commands have a friendlier mnemonic format, and refer to objects by user-defined names rather than id numbers. Some of the names are declared by commands, while others are taken from description files. While processing a command, HIP transparently refers to the current Physical Network Description (PND) and to Process Type Description (PTD) files, both of which contain declarations of user-defined names. A single HIP command may result in the generation of several commands to Process Manager.

An explanation and example are provided for the first command. The formats for a number of other selected commands are listed. The nature of many commands is evident from the leading keyword. Modules of commands supporting activities other than process management may also be added to HIP.

ALLOCATE <newProcessName> <newProcessDescriptionFileName> <stationName>

Explanation: Allocates a new process to the specified station. The process will be referred to by the user defined name <newProcessName>. Parameters for the process are looked up from the specified process type description (PTD) file. <stationName> must be present in the currently selected PND. Declarations in the PTD include the number of input queues and outlets for the process, as well as the lengths and policies of each input queue. The PTD also declares memory requirements of the process. ALLOCATE reserves appropriately sized memory regions for the process. (The PTD also contains other information as well. This includes user-defined names for referring to each input queue and outlet of the process, used in the CONNECT command. It also includes object file names, which are used by the LOAD command.)

Example: allocate master vnd/master.idf hub

Allocates a new process called 'master' to the station 'hub'. Information about master is looked up from the PTD file named vnd/master.idf. Appropriately sized memory regions are reserved for master.

ALLOCATEPID <newProcessName> <newProcessDescriptionFileName> <newPid> <stationName>

CLEARALL

CONNECT <sourceProcessName> <outletName> <destinationProcessName> <inputQueueName> <route>

CONNECTIONREPORT <processName>

CONTINUE <processName>

LOAD <processName>

PROCESSINFO <processName>

PROCESSLIST

PROCESSLISTAT <stationName>

PROCESSREPORT <processName>

REMOVE <processName>

SELECTPND <newPhysicalNetworkDescriptionFileName>

SETPRIORITY <processName> <newPriority>

START <processName>

STATIONINFO <stationName>

STATIONLIST

STATIONREPORT <stationName>

SUSPEND <processName>

Appendix C Demonstration Programs for NetKernel

The NetKernel programs written to date have been fairly simple tests and demonstrations of the kernel and development tools. As such, they are quite trivial in nature. Several of these programs and a few performance measurements are presented.

These programs informally demonstrate the operational requirements of Section 2.2. The processes communicate and synchronize with one another (requirement B1). The mapping of a program's software network to a particular configuration of hardware can be easily changed (requirement B2) by modifying the Mapping Description (MD). Through the Physical Network Description (PND), a program can be mapped to different configurations of hardware.

The results of this appendix were collected using M68000 stations interconnected by 9600 baud serial links (i.e. the hardware described in Section 3.3).

C.1 Ring Program and Message System Performance

In this program, a token circulates around processes arranged in a ring. This is illustrated in Figure 30 for a ring consisting of a master and two slaves. Any number of slaves may be used. In the case of zero slaves, the outlet of master is simply connected back to an input queue of master.

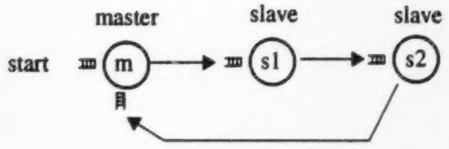


Figure 30 Software network for a ring consisting of a master and two slaves

To begin circulation of a token, a start message is sent interactively to the *start* queue of master. The message should contain the number of times that the token is to be circulated.

Master then forms a token message containing the number and sends it through its outlet. Each slave simply forwards any message it receives through its outlet. The token is decremented each time it returns to master. When the token reaches zero, master stops forwarding it and sends a completion reply to the original start message. The reply indicates that the token has been successfully circulated the requested number of times. It is interesting to note that multiple tokens may circulate around a ring at the same time if several start commands are sent in rapid succession.

This program has been used to measure the performance of our implementation of NetKernel. A ring consisting of master and a single slave was used. The interval between sending the start command and receiving the completion reply was timed for a large number of circulations. The throughput for local messages was determined by mapping both processes to the same station. The throughput for messages over a single serial link was determined by mapping the processes to two different stations connected by a serial link. One circulation of the token counted as two messages since a two process ring was used. Results are shown in Table 7 for two different sizes of message. Case *b* differs in that master and slave request that the token buffer be copied upon Send(), rather than allowing its ownership to be transferred. In addition to data bytes, the link and network layer headers add 32+2N header bytes for messages transmitted over serial links. N is the number of hops the message traverses, with N=1 for case *c*.

	Throughput (messages/second)	
	4 bytes of data	180 bytes of data
a. Local	176	176
b. Local with copy (and call to Free())	130	130
c. Nonlocal (one hop over a serial link)	13	3.8

Table 7 Message throughput as measured using a ring of two processes

No detailed analysis of performance will be made, but we will quickly show that these values are reasonable for our hardware. In case *a*, four system calls (Send(), Wait(), Take(), and Free() the reply key) and one context switch are being made per message exchange. So a performance of 704 system calls/second and 176 context switches/second is obtained. The M68000 units execute approximately 500 000 machine instructions per second (0.5 MIPs). Dividing this by (704+176) results in an estimate of 568 machine instructions for each system call and context switch, which is a reasonable figure. In case *b*, an extra system call per message is performed to Free() the token buffer and the Send() call must internally allocate a new buffer every time it is called. For case *c*, the link performance needs to be considered. The link packet sizes are 38 and 214 bytes for the cases of 4 and 180 data bytes, respectively. The throughputs obtained are 494 and 813 bytes/second, respectively. A 9600 baud link with one start bit, eight data bits, and one stop bit has a maximum throughput of 960 characters per second. So link utilizations of 51% and 85%

are being obtained, again reasonable figures. An additional 6 characters per message for link protocol signaling (2 characters for each of START, END, and ACK) were omitted from the calculations.

Several files from this example are included in Appendix D.

C.2 Synthetic Program with Mapping Examples

A simple synthetic program, referred to as SYNTH, has been written to demonstrate different mappings of a software network. A synthetic program is one which performs no useful computation, but demonstrates some capabilities of a system or serves as a basis for comparison between cases [Poplawski91]. The program presented here demonstrates that the mapping of a concurrent program can affect its performance, and that NetKernel is capable of achieving speedup for this particular program.

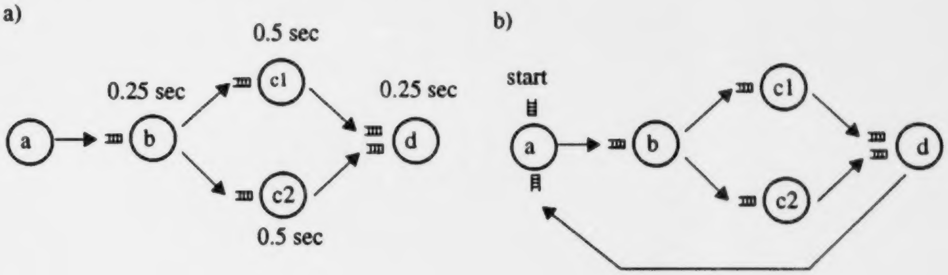
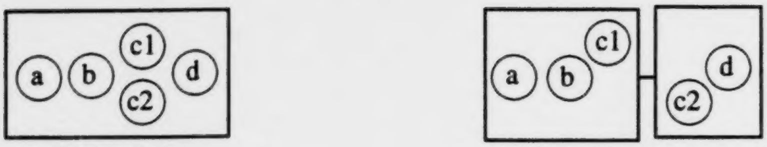


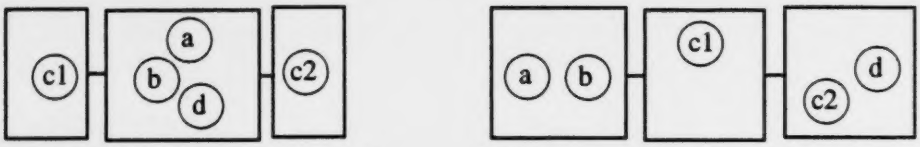
Figure 31 Software network for SYNTH
a) Basic software network b) Software network with feedback loop added to simplify measurement

The basic software network is shown in Figure 31a. Process *a* is a source of input data (in a real situation, process *a* might read data from some device). For each input message generated by process *a*, several computations and messages result as follows. Process *a* sends the messages it generates to *b*. For each message received, process *b* performs 0.25 seconds of computation and then sends two messages, one to *c1* and one to *c2*. Processes *c1* and *c2* are identical, although they are connected differently in the software network. For each message received, these processes perform 0.50 seconds of computation and then send a message to process *d*. Process *d* waits for the two messages from *c1* and *c2*, and then performs 0.25 seconds of computation. It then discards the messages (in a real situation, process *d* might send data to some output device). All messages contain 4 data bytes.

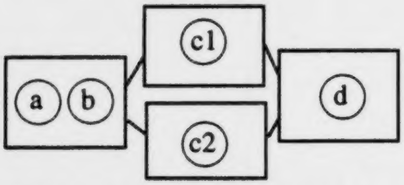
It should be stated at the outset that the parameter values used in SYNTH were selected such that the overhead time in running the distributed version is low compared to the computation time. This ensures that the performance will improve when more processors are used, up to a limit dependent on the amount of inherent parallelism in SYNTH. By select-



A) Single station B) Two stations arranged in a line



C) Three stations arranged in a line D) Two stations arranged in a line



E) Four stations arranged in a ring

Figure 32 Five mappings of SYNTH to four different physical networks

ing different parameter values (i.e. smaller time constants), it would be possible to come up with a version of SYNTH where the best performance is on a single processor. The overhead of nonlocal message-passing would outweigh the benefits of performing the (shorter) computations on different stations. The selection of a good mapping must consider both the benefits of parallel computation and the overheads involved in distributing the computation. Due to its flexibility of mapping, NetKernel provides a useful tool for the verification of experimental techniques to select good mappings.

In order to determine the maximum throughput possible for different mappings, a feedback loop was added as shown in Figure 31b. Process *a* sends N messages initially, and then sends an additional message each time an acknowledgment is received from process *d*. This is a window scheme for flow control. It has the effect of limiting the rate at which process *a* produces messages so that it will not produce the (k+N)th message until input message k has completely propagated through the software network. A window size of N=5 was used. The program is operated by sending a message to the start queue. The time for 30 messages from process *a* to propagate through the software network was measured. Results were recorded for five different mappings involving four different physical networks, illustrated in Figure 32. Results for each mapping are given in Table 8.

Mapping	Ideal T	Actual T	Actual Throughput= 30/T
a	45	49	0.61
b	23	26	1.2
c	15	19	1.6
d	23	26	1.2
e	16	19	1.6
T = total time for 30 input messages (sec) (All values for T are rounded to the nearest second)			

Table 8 Throughput measurements for various mappings of SYNTH

The column labelled *Actual T* shows that the performance of SYNTH varies significantly for different physical networks and mappings. Mappings *c* and *e* provide the highest performance, with *c* being preferable since it uses fewer stations. Results for *c* and *d* illustrate that even for the same physical network, the mapping has an impact on performance.

The column labelled *Ideal T* shows the ideal value for T for with no overhead. Ideal T considers only the processing times shown in Figure 31 (which were set up using empty 'for' loops). In practice, there is overhead time for message passing, context switching, etc. Ideal T can be computed for SYNTH in a fashion similar to computing the execution time for a set of data to flow through a pipeline. Mapping *a* is trivial since only one station is involved. For every input message, 0.25+0.5+0.5+0.25 = 1.5 seconds of computation must be performed by the station. For 30 input messages, the total time is (30)(1.5) = 45 sec, which is close to the measured T of 49 seconds. For mapping *b*, the processes have been mapped to two stations. While process *c* operates on the first message, station 2 will be idle for 0.25 seconds. The remaining component of ideal T is attributed to station 2 operating on 30 messages. So the complete expression for mapping *b* is ideal T = 0.25 + (30)(0.5+0.25) = 22.75. The factor (0.5+0.25) is equal to the amount of time station 2 (i.e. processes c2 and d) spends on each message. Ideal T for the other mappings is computed similarly. It can be seen that the actual T for mappings *b-e* is significantly better than the ideal T of the single processor case (mapping *a*). This indicates that the parallel version provides speedup over the best possible single processor implementation.

C.3 Repulse Game

A simple game has been developed to test the NetKernel runtime environment. To summarize the game's operation, one to eight balls are released onto a rectangular playing field. The field is surrounded by walls. The balls and walls all repel each other. The repulsiveness of the balls and walls can be controlled by the player, along with the positions of balls. A "graphical" representation of the playing field and balls is displayed on a VT100 terminal.

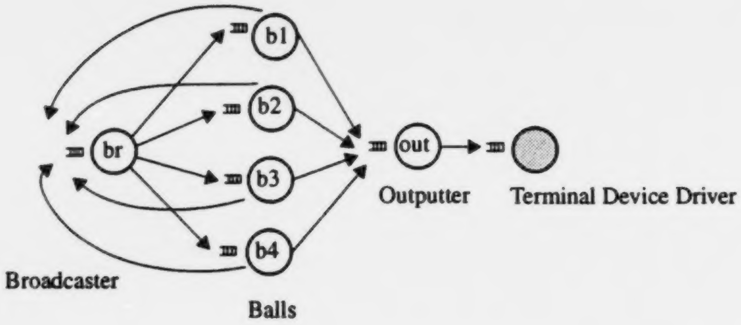


Figure 33 Software network for Repulse game

Figure 33 shows the software network for a game with four balls. A process is associated with each of the four balls (processes *b1* through *b4*). Each ball maintains its own internal model of the playing field and other balls. The game evolves in real time, with each ball having its own time constant. Periodically, each ball process computes a new position for itself based on its internal model. All balls need to be informed whenever a ball moves, but NetKernel has no broadcast operation. So a Broadcaster process was used. Whenever *br* receives a message, it sends a copy of the message to all balls. Each time a ball moves, that ball also sends an output message to Outputter, which forwards the message to Terminal Device Driver to update the display. The processes wait for their period to expire using a timer (connections to Timer were omitted from Figure 33 for simplicity).

To demonstrate a mixed host/target environment, a modified version of Repulse was developed. The outlet from Outputter is instead connected to a process 3dField executing on a Next workstation under the prototype kernel (Section 3.8.5). 3dField displays a 3 dimensional representation of the playing field and balls within a window.

Appendix D Example Source Code and Description Files

This appendix contains listings of several files for the Ring example of Section C.1. A Software Network Description (SND), Physical Network Description (PND), and Mapping Description (MD) are included. The source code and Process Type Description (PTD) for the slave process are also included.

The script which Script Generator (SG) generates from the SND, PND, and MD is presented last. This script is read by the Host Interface Program (HIP) to load and start the software network.

D.1 SND, PND, MD

```
// ring2.snd: software network description(SND) for a ring containing one master and one slave
// This file declares the processes of the software network and their interconnection
NEWPROCESS Master vnd/master.idf; {
  OUTLET OutputOut OutputTerminal InputIn;
  OUTLET ToNextOut Slave FromPreviousIn;
}
NEWPROCESS Slave vnd/slave.idf; {
  OUTLET OutputOut OutputTerminal InputIn;
  OUTLET ToNextOut Master FromPreviousIn;
}
SERVICEPROCESS OutputTerminal TerminalServiceType; {
  OUTLET OutputOut Master CommandIn;
}
```

```
// hubsat.pnd: physical network description (PND) for a network of two stations joined by a serial link
// The stations are referred to as hub and sat. A terminal is attached to hub.
STATION Hub vme68; {
  ROUTE Hub;
}
STATION Sat vme68; {
  ROUTE Hub Sat;
}
```

```
LINK . Hub 0B Sat 09;
UNIQUESERVICE Hub_Terminal_Service TerminalServiceType; {
 STATION Hub;
 PID 10;
 QUEUE InputIn 04;
 OUTLET OutputOut 00;
}
```

```
// ring2_to_hubsat.map: mapping description (MD) to map ring2.snd to hubsat.pnd
// Process master is allocated to station sat. Process slave is allocated to station hub.
NEWPROCESS Master Sat;
NEWPROCESS Slave Hub;
SERVICEPROCESS OutputTerminal Hub_Terminal_Service;
ROUTE Master OutputOut Sat Hub;
ROUTE Master ToNextOut Sat Hub;
ROUTE Slave OutputOut Hub;
ROUTE Slave ToNextOut Hub Sat;
ROUTE OutputTerminal OutputOut Hub Sat;
```

D.2 slave.c, PTD for slave

```
// slave.c: source code of the slave process
#include "MP0.h"
#define PREVOUS_IN_MASK 2
#define OUTPUT_OUT 0
#define NEXT_OUT 1

void show(int code, char *message) {
    int messageNumber;
    if (code)
        MP0_sendOutlet(OUTPUT_OUT, (void *)message, strlen(message)+1, MP0_NOFREE,
MP0_NOREPLY, MP0_NOCONFIRM, &messageNumber);
}

void main() {
    int ret;
    int size,qid,messagenumber,replyflag;
    void *dptr,*hptr;

    ret = MP0_init();
    show(1, "***** Process SLAVE is running...\n");
    while(1) {
        ret = MP1_wait_take(PREVIOUS_IN_MASK, MP0_TAKE_ANY, MP1_NOTIMEOUT,
MP1_SYSTEM_IN, &hptr, &dptr, &size, &qid, &messagenumber, &replyflag);
```

```
        show(ret,"slave:MP1_wait_take, Q1");
        ret = MP0_sendOutlet(NEXT_OUT, dptr, size, MP0_FREE, MP0_NOREPLY,
MP0_NOCONFIRM, &messagenumber);
        show(ret,"slave:MP0_sendOutlet");
         ret = MP0_free((void *) hptr);
        show(ret,"slave:MP0_free, hptr");
    }
}
```

```
// slave.tdf: process type description (PTD) for slave
// This file declares input queue names, outlet names, memory regions lengths, object file names, priority.
// This file also refers to default.df, which contains a number of default settings.
#SUPER_FILE df/default.df;

NUMBER_INPUT_QUEUES 02;
QUEUE SystemIn    00;
QUEUE FromPreviousIn  01;

NUMBER_OUTLETS 02;
OUTLET OutputOut 00;
OUTLET ToNextOut01;

SHARED_CODE_SIZE  1000;
PRIVATE_DATA_SIZE  600;
STACK_SIZE       0500;
START_CODE_REGION shared;
SHARED_CODE_FILE  process/slave/slave.shared.mot;
PRIVATE_DATA_FILE process/master/master.unshared.mot;
PRIORITY 2;
```

D.3 Script for HIP Generated by SG

```
// ring2_to_hubsat.hip: Loads and starts ring2.snd as described by the MD ring2_to_hubsat.map
ALLOCATE master vnd/master.idf sat;
ALLOCATE slave vnd/slave.idf hub;
CONNECT master outputout hub_terminal_service inputin sat hub;
CONNECT master tonextout slave frompreviousin sat hub;
CONNECT slave outputout hub_terminal_service inputin hub;
CONNECT slave tonextout master frompreviousin hub sat;
CONNECT hub_terminal_service outputout master commandin hub sat;
LOAD master;
LOAD slave;
START master;
START slave;
```

Appendix E Event Log Report Showing Clock Servicing

NetKernel provides an option for recording timestamped events into an event log, as described in Section 3.5.8.1. Some standard system events are predefined. The user may define additional events by inserting traps into user code. An example of a log will be presented and some observations concerning clock servicing will be drawn from it.

The Event Log Report Generator (Section 4.5) accepts a hexadecimal memory image of the event log. It is executed off-line on a workstation to produce a formatted textual report of the event log. An excerpt from an example report is shown in Figure 34. A ring of two processes (similar to Figure 30, but containing only one slave) was executing on a single station during this trace. Most of the standard events were disabled. Only three events were enabled: context switches, entry to clock service routine, and exit from clock service routine.

The TIME TO PREVIOUS SAME column indicates the elapsed time since most recent event of the same type. This provides a convenient measure of the interval between context switches, for instance. The TIME TO PREVIOUS indicates the elapsed time since the immediately preceding event. The clock entry and clock exit events are a matching pair, with the time in square brackets indicating the elapsed time from entry to exit. Matching pairs are also used to measure the duration of system calls, for instance.

Pids 40 and 41 are the master and slave processes, respectively. The '???' indicates that ELRG does not know the names of these processes, although it may be customized to include that information. An estimate of the overhead for clock servicing may be made from the log. As discussed in Section 3.5.7, the clock interrupt rate is 200/second, but most service activities are performed on only every 10th interrupt. This is evident on the log: The first and eleventh clock interrupts take longer to service than the others. The bracketed service times for the clock interrupts result in an estimate of clock overhead equal to 4.2% of cpu time. It is interesting to examine the skewing of clock interrupts, which should otherwise arrive every 5000 microseconds. The TIME TO PREV SAME on TICK ENTRY lines shows the interval between clock interrupts. The skewing is due to critical sections within the system calls being made by master and slave (which are making system calls in rapid succession since they perform no significant computation). Nonetheless, no clock interrupts were lost in Figure 34 since it shows 11 clock interrupts over a

```
2048 events in trace.
Trace spans 2581075 microseconds, approx. 2.58 sec. or 0 min. 2 sec.
All times are in microseconds. Timestamp resolution is 25 microseconds.
Each log entry incurs approx. 100 microseconds overhead.
-----   -------        -----  ---------   --------   ------------------------
 EVENT  RELATIVE        RAW    TIME TO    TIME TO   DESCRIPTION
   NUM      TIME       TIME  PREV SAME   PREVIOUS   [TIME TO EXIT IF MATCHED]
 -----  --------       ----  ---------   --------   ------------------------
[ ... deleted ...]
   50)     89975  262112875       6975        950   CONTEXT SWITCH to pid 0x41 (???)
   51)     94850  262108000       6000       4875      TICK ENTRY
   52)     95375  262107475       6350        525      TICK EXIT [525]
   53)     96975  262105875       7000       1600   CONTEXT SWITCH to pid 0x40 (???)
   54)     98325  262104525       3475       1350      TICK ENTRY
   55)     98500  262104350       3125        175      TICK EXIT [175]
   56)    102950  262099900       4625       4450      TICK ENTRY
   57)    103125  262099725       4625        175      TICK EXIT [175]
   58)    103950  262098900       6975        825   CONTEXT SWITCH to pid 0x41 (???)
   59)    108825  262094025       5875       4875      TICK ENTRY

   60)    109000  262093850       5875        175      TICK EXIT [175]
   61)    110600  262092250       6650       1600   CONTEXT SWITCH to pid 0x40 (???)
   62)    116125  262086725       7300       5525      TICK ENTRY
   63)    116300  262086550       7300        175      TICK EXIT [175]
   64)    117250  262085600       6650        950   CONTEXT SWITCH to pid 0x41 (???)
   65)    117950  262084900       1825        700      TICK ENTRY
   66)    118125  262084725       1825        175      TICK EXIT [175]
   67)    123125  262079725       5175       5000      TICK ENTRY
   68)    123275  262079575       5150        150      TICK EXIT [150]
   69)    124225  262078625       6975        950   CONTEXT SWITCH to pid 0x40 (???)

   70)    129750  262073100       6625       5525      TICK ENTRY
   71)    129900  262072950       6625        150      TICK EXIT [150]
   72)    130850  262072000       6625        950   CONTEXT SWITCH to pid 0x41 (???)
   73)    135725  262067125       5975       4875      TICK ENTRY
   74)    135900  262066950       6000        175      TICK EXIT [175]
   75)    137500  262065350       6650       1600   CONTEXT SWITCH to pid 0x40 (???)
   76)    138175  262064675       2450        675      TICK ENTRY
   77)    138350  262064500       2450        175      TICK EXIT [175]
   78)    143375  262059475       5200       5025      TICK ENTRY
   79)    143875  262058975       5525        500      TICK EXIT [500]
[... deleted ...]
```

Figure 34 Example output from Event Log Report Generator

span of 54525 microseconds. Lost interrupts up to a maximum of nine in a row are detected and recorded by the microkernel, appearing occasionally. For correct timekeeping, only one in ten interrupts need actually arrive, so there is little danger of any timers falling behind.

The recording of a log entry is fairly expensive. By generating two log entries one after the other, the overhead per entry was estimated to be approximately 100 microseconds (which agrees with adding up the execution time for the machine instructions involved). So the estimate for clock overhead made above is substantially greater than the overhead incurred if clock entry and exit logging are turned off (the usual case), which works out to 2.2% of cpu time.